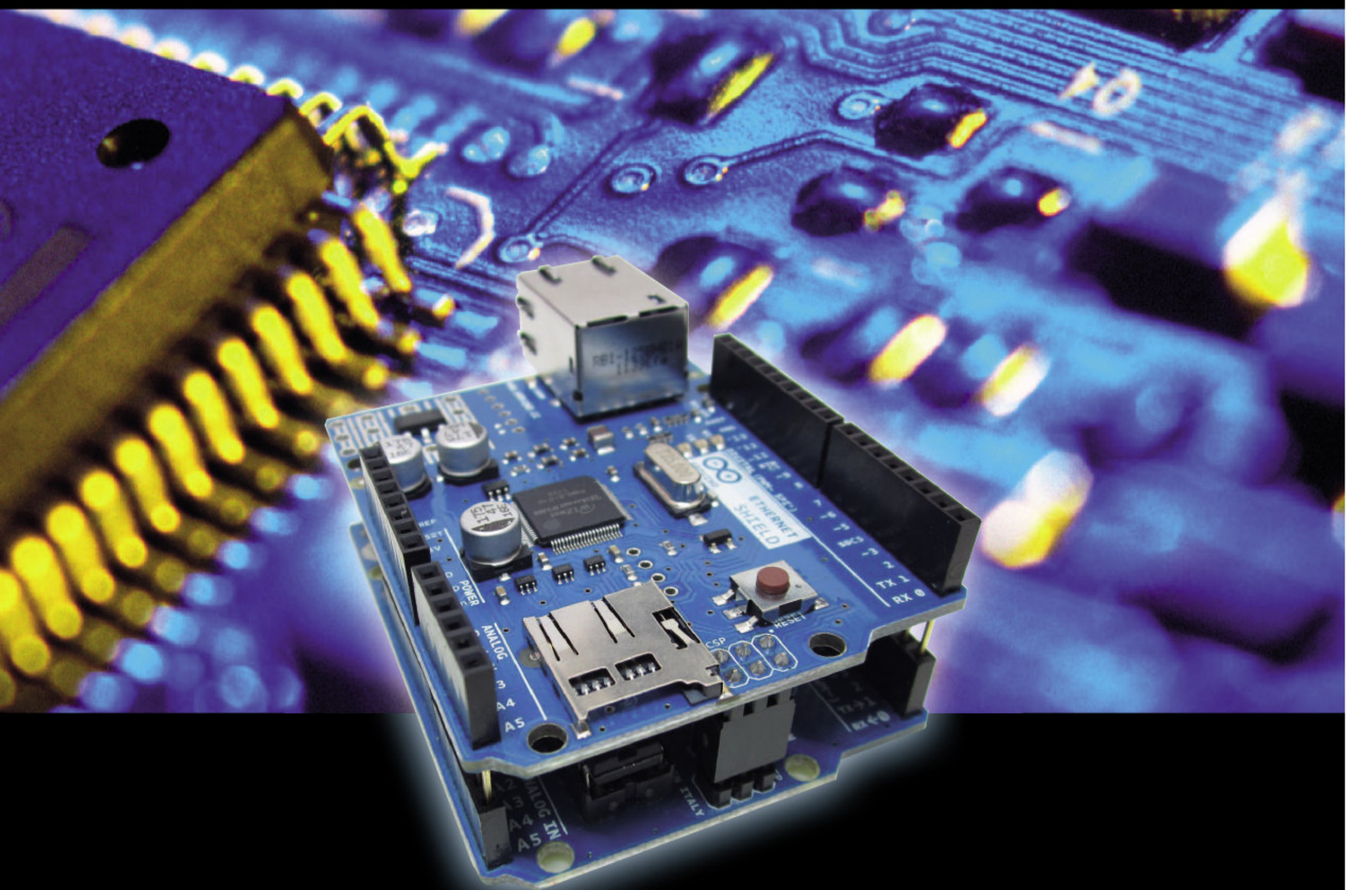
ARDUINO

Curso práctico de formación



Óscar Torrente Artero

Alfaomega

libros RC

ARDUINO

Curso práctico de formación

ARDUINO

Curso práctico de formación

Óscar Torrente Artero





Los esquemas eléctricos han sido realizados con CircuitLab: http://www.circuitlab.com
Los gráficos de circuitos han sido realizados con Fritzing: http://www.fritzing.org
Los retoques han sido realizados con Inkscape y Gimp: http://inkscape.org, http://gimp.org
Las imágenes han sido obtenidas por medios propios o bien descargadas de la Wikipedia o Ladyada.net (con licencia CC-Share-Alike): http://es.wikipedia.org, http://www.ladyada.net

Datos catalográficos
Torrente, Óscar
Arduino. Curso práctico de formación
Primera Edición
Alfaomega Grupo Editor, S.A. de C.V., México
ISBN: 978-607-707-648-3
Formato: 17 x 23 cm Páginas: 588

Arduino. Curso práctico de formación
Óscar Torrente Artero
ISBN: 978-84-940725-0-5 edición original publicada por RC Libros, Madrid, España

Primera edición: Alfaomega Grupo Editor, México, febrero 2013

Pitágoras 1139, Col. Del Valle, 03100, México D.F.

Miembro de la Cámara Nacional de la Industria Editorial Mexicana
Registro No. 2317

Pág. Web: **http://www.alfaomega.com.mx**
E-mail: **atencionalcliente@alfaomega.com.mx**

ISBN: 978-607-707-648-3

Impreso en México. Printed in Mexico.

Empresas del grupo:
México: Alfaomega Grupo Editor, S.A. de C.V. – Pitágoras 1139, Col. Del Valle, México, D.F. – C.P. 03100. Tel.: (52-55) 5575-5022 – Fax: (52-55) 5575-2420 / 2490. Sin costo: 01-800-020-4396 E-mail: atencionalcliente@alfaomega.com.mx
Colombia: Alfaomega Colombiana S.A. – Carrera 15 No. 64 A 29, Bogotá, Colombia, Tel.: (57-1) 2100122 – Fax: (57-1) 6068648 – E-mail: cliente@alfaomega.com.co
Chile: Alfaomega Grupo Editor, S.A. – General del Canto 370, Providencia, Santiago, Chile Tel.: (56-2) 947-9351 – Fax: (56-2) 235-5786 – E-mail: agechile@alfaomega.cl
Argentina: Alfaomega Grupo Editor Argentino, S.A. – Paraguay 1307 P.B. Of. 11, C.P. 1057, Buenos Aires, Argentina, – Tel./Fax: (54-11) 4811-0887 y 4811 7183 – E-mail: ventas@alfaomegaeditor.com.ar

A mi madre

ÍNDICE

INTRODUCCIÓN

A quién va dirigido este libro

Construir coches y helicópteros teledirigidos, fabricar diferentes tipos de robots inteligentes, crear sintetizadores de sonidos, montar una completa estación meteorológica (con sensores de temperatura, humedad, presión...), ensamblar una impresora 3D, monitorizar la eficacia de nuestro refrigerador de cervezas desde el jardín, controlar a través de Internet la puesta en marcha de la calefacción y de las luces de nuestra casa cuando estemos lejos de ella, enviar periódicamente los datos de consumo doméstico de agua a nuestra cuenta de Twitter, diseñar ropa que se ilumine ante la presencia de gas, establecer un sistema de secuencia de golpes a modo de contraseña para abrir puertas automáticamente, apagar todos los televisores cercanos de una sola vez, implementar un sistema de riego automático y autorregulado según el estado de humedad detectada en la tierra, elaborar un theremin de rayos de luz, fabricar un reloj-despertador musical, utilizar una cámara de vídeo como radar para recibir alarmas de intrusos en nuestro teléfono móvil, jugar al tres en raya mediante órdenes habladas, etc. Todo lo anterior y muchísimo más se puede conseguir con Arduino.

Este libro está dirigido, pues, a todo aquel que quiera investigar cómo conectar el mundo físico exterior con el mundo de la electrónica y la informática, para lograr así una interacción autónoma y casi "inteligente" entre ambos mundos. Ingenieros, artistas, profesores o simples aficionados podrán conocer las posibilidades que les ofrece el ecosistema Arduino para llevar a cabo casi cualquier proyecto que la imaginación proponga.

Este curso está pensado para usuarios con nulos conocimientos de programación y de electrónica. Se presupone que el lector tiene un nivel básico de informática doméstica (por ejemplo, sabe cómo descomprimir un archivo “zip” o cómo crear un acceso directo) pero no más. Por lo tanto, este texto es ideal para todo aquel que no haya programado nunca ni haya realizado ningún circuito eléctrico. En cierto sentido, gracias a la “excusa” de Arduino, lo que tiene el lector en sus manos es un manual de iniciación tanto a la electrónica como a la programación básica.

El texto se ha escrito facilitando al lector autodidacta una asimilación gradual de los conceptos y procedimientos necesarios para ir avanzando poco a poco y con seguridad a lo largo de los diferentes capítulos, desde el primero hasta el último. Esta estructura hace que el texto también pueda ser utilizado perfectamente como libro de referencia para profesores que impartan cursos de Arduino dentro de diversos ámbitos (educación secundaria, formación profesional, talleres no reglados, etc.). Aderezado con multitud de ejemplos de circuitos y códigos, su lectura permite la comprensión del universo Arduino de una forma práctica y progresiva.

No obstante, aunque muy completo, este curso no es una referencia o compendio exhaustivo de todas las funcionalidades que ofrece el sistema Arduino. Sería imposible abarcarlas todas en un solo volumen. El lector experimentado notará que en las páginas siguientes faltan por mencionar y explicar aspectos avanzados tan interesantes (algunos de los cuales pueden dar lugar a un libro entero por sí mismos) como el papel de Arduino en la construcción de robots o de impresoras 3D, o las posibilidades de comunicación entre Arduino y dispositivos con sistema Android, por ejemplo.

Cómo leer este libro

Este curso se ha escrito teniendo en cuenta varios aspectos. Se ha procurado en la medida de lo posible escribir un manual que sea **autocontenido** y **progresivo.** Es decir, que no sea necesario recurrir a fuentes de información externas para comprender todo lo que se explica, sino que el propio texto sea autoexplicativo en sí mismo. Y además, que toda la información expuesta sea mostrada de forma ordenada y graduada, sin introducir conceptos o procedimientos no explicados con anterioridad. Por tanto, se recomienda una lectura secuencial, desde el primer capítulo hasta el último, sin saltos.

La metodología utilizada en este texto se basa fundamentalmente en la exposición y explicación pormenorizada de multitud de ejemplos de código **cortos y**

concisos: se ha intentado evitar códigos largos y complejos, que aunque interesantes y vistosos, pueden distraer y desorientar al lector al ser demasiado inabarcables. La idea no es presentar proyectos complejos ya acabados, sino exponer de la forma más simple posible los conceptos básicos. En este sentido, se aportan multitud de enlaces para ampliar los conocimientos que no tienen espacio en el libro: muchos son los temas que se proponen (electricidad, electrónica, algoritmia, mecánica, acústica, electromagnetismo, etc.) para que el lector que tenga iniciativa pueda investigar por su cuenta.

La estructura de los capítulos es la siguiente: el primer capítulo introduce los conceptos básicos de electricidad en circuitos electrónicos, y describe –mediante ejemplos concretos– el comportamiento y la utilidad de los componentes presentes en la mayoría de estos circuitos (como pueden ser las resistencias, condensadores, transistores, placas de prototipado, etc.). El segundo capítulo expone las diferentes placas que forman el ecosistema Arduino, los componentes que las forman y los conceptos más importantes ligados a esta plataforma. El tercer capítulo muestra el entorno de programación oficial de Arduino y describe su instalación y configuración. El cuarto capítulo repasa la funcionalidad básica del lenguaje de programación Arduino, proponiendo múltiples ejemplos donde se pueden observar las distintas estructuras de flujo, funciones, tipos de datos, etc., empleados por este lenguaje. El quinto capítulo muestra la diversidad de librerías oficiales que incorpora el lenguaje Arduino, y aprovecha para profundizar en el manejo del hardware que hace uso de ellas (tarjetas SD, pantallas LCD, motores, etc.). El sexto capítulo se centra en el manejo de las entradas y salidas de la placa Arduino, tanto analógicas como digitales, y su manipulación a través de pulsadores o potenciómetros, entre otros. El séptimo capítulo explica varios tipos de sensores mediante ejemplos de cableado y código; entre los sensores tratados encontramos sensores de luz, infrarrojos, de distancia, de movimiento, de temperatura, de humedad, de presión atmosférica, de fuerza y flexión, de sonido... El octavo y último capítulo analiza la capacidad que tienen las placas Arduino para comunicarse con otros dispositivos (como computadores, teléfonos móviles u otras placas Arduino) mediante redes TCP/IP cableadas o inalámbricas (Wi-Fi), y mediante Bluetooth, además de proponer multitud de ejemplos prácticos de interacción y transmisión de datos a través de la red.

1 ELECTRÓNICA BÁSICA

CONCEPTOS TEÓRICOS SOBRE ELECTRICIDAD

¿Qué es la electricidad?

Un electrón es una partícula subatómica que posee carga eléctrica negativa. Por lo tanto, debido a la ley física de atracción entre sí de cargas eléctricas de signo opuesto (y de repulsión entre sí de cargas eléctricas de mismo signo), cualquier electrón siempre es atraído por una carga positiva equivalente.

Una consecuencia de este hecho es que si, por razones que no estudiaremos, en un extremo (también llamado "polo") de un material conductor aparece un exceso de electrones y en el otro polo aparece una carencia de estos (equivalente a la existencia de "cargas positivas"), los electrones tenderán a desplazarse a través de ese conductor desde el polo negativo al positivo. A esta circulación de electrones por un material conductor se le llama "electricidad".

La electricidad existirá mientras no se alcance una compensación de cargas entre los dos polos del conductor. Es decir, a medida que los electrones se desplacen de un extremo a otro, el polo negativo será cada vez menos negativo y el polo positivo será cada vez menos positivo, hasta llegar el momento en el que ambos extremos tengan una carga global neutra (es decir, estén en equilibrio). Llegados a esta situación, el movimiento de los electrones cesará. Para evitar esto, en la práctica se suele utilizar una fuente de alimentación externa (lo que se llama un "generador") para restablecer constantemente la diferencia inicial de cargas entre los extremos del

conductor, como si fuera una "bomba". De esta manera, mientras el generador funcione, el desplazamiento de los electrones podrá continuar sin interrupción.

¿Qué es el voltaje?

En el estudio del fenómeno de la electricidad existe un concepto fundamental que es el de voltaje entre dos puntos de un circuito eléctrico (también llamado "tensión", "diferencia de potencial" o "caída de potencial"). Expliquémoslo con un ejemplo.

Si entre dos puntos de un conductor no existe diferencia de cargas eléctricas, el voltaje entre ambos puntos es cero. Si entre esos dos puntos aparece un desequilibrio de cargas (es decir, que en un punto hay un exceso de cargas negativas y en el otro una ausencia de ellas), aparecerá un voltaje entre ambos puntos, el cual será mayor a medida que la diferencia de cargas sea también mayor. Este voltaje es el responsable de la generación del flujo de electrones entre los dos puntos del conductor. No obstante, si los dos puntos tienen un desequilibrio de cargas entre sí pero están unidos mediante un material no conductor (lo que se llama un material "aislante"), existirá un voltaje entre ellos pero no habrá paso de electrones (es decir, no habrá electricidad).

Generalmente, se suele decir que el punto del circuito con mayor exceso de cargas positivas (o dicho de otra forma: con mayor carencia de cargas negativas) es el que tiene el "potencial" más elevado, y el punto con mayor exceso de cargas negativas es el que tiene el "potencial" más reducido. Pero no olvidemos nunca que el voltaje siempre se mide entre dos puntos: no tiene sentido decir "el voltaje en este punto", sino "el voltaje en este punto respecto a este otro"; de ahí sus otros nombres de "diferencia de potencial" o "caída de potencial".

Así pues, como lo que utilizaremos siempre serán las diferencias de potencial relativas entre dos puntos, el valor numérico absoluto de cada uno de ellos lo podremos asignar según nos convenga. Es decir, aunque 5, 15 y 25 son valores absolutos diferentes, la diferencia de potencial entre un punto que vale 25 y otro que vale 15, y la diferencia entre uno que vale 15 y otro que vale 5 da el mismo resultado. Por este motivo, y por comodidad y facilidad en el cálculo, al punto del circuito con potencial más reducido (el de mayor carga negativa, recordemos) se le suele dar un valor de referencia igual a 0.

También por convenio (aunque físicamente sea en realidad justo al contrario) se suele decir que la corriente eléctrica va desde el punto con potencial mayor hacia

otro punto con potencial menor (es decir, que la carga acumulada en el extremo positivo es la que se desplaza hacia el extremo negativo).

Para entender mejor el concepto de voltaje podemos utilizar la analogía de la altura de un edificio: si suponemos que el punto con el potencial más pequeño es el suelo y asumimos este como el punto de referencia con valor 0, a medida que un ascensor vaya subiendo por el edificio irá adquiriendo más y más potencial respecto el suelo: cuanta más altura tenga el ascensor, más diferencia de potencial habrá entre este y el suelo. Cuando estemos hablando de una "caída de potencial", querremos decir entonces (en nuestro ejemplo) que el ascensor ha disminuido su altura respecto al suelo y por tanto tiene un voltaje menor.

La unidad de medida del voltaje es el voltio (V), pero también podemos hablar de milivoltios (1 mV = 0,001 V), o de kilovoltios (1 kV = 1000 V). Los valores típicos en proyectos de electrónica casera como los que abordaremos en este libro son de 1,5 V, 3,3 V, 5 V... aunque cuando intervienen elementos mecánicos (como motores) u otros elementos complejos, se necesitará aportar algo más de energía al circuito, por lo que los valores suelen algo mayores: 9 V, 12 V o incluso 24 V. En todo caso, es importante tener en cuenta que valores más allá de 40 V pueden poner en riesgo nuestra vida si no tomamos las precauciones adecuadas; en los proyectos de este libro, de todas formas, no se utilizarán nunca voltajes de esta magnitud.

¿Qué es la intensidad de corriente?

La intensidad de corriente (comúnmente llamada "corriente" a secas) es una magnitud eléctrica que se define como la cantidad de carga eléctrica que pasa en un determinado tiempo a través de un punto concreto de un material conductor. Podemos imaginar que la intensidad de corriente es similar en cierto sentido al caudal de agua que circula por una tubería: que pase más o menos cantidad de agua por la tubería en un determinado tiempo sería análogo a que pase más o menos cantidad de electrones por un cable eléctrico en ese mismo tiempo.

Su unidad de medida es el amperio (A), pero también podemos hablar de miliamperios (1 mA = 0,001 A), de microamperios (1 µA = 0,001 mA), o incluso de nanoamperios (1 nA = 0,001 µA).

Tal como ya hemos comentado, se suele considerar que en un circuito la corriente fluye del polo positivo (punto de mayor tensión) al polo negativo (punto de menor tensión) a través de un material conductor.

¿Qué es la corriente continua (DC) y la corriente alterna (AC)?

Hay que distinguir dos tipos fundamentales de circuitos cuando hablamos de magnitudes como el voltaje o la intensidad: los circuitos de corriente continua (o circuitos DC, del inglés "Direct Current") y los circuitos de corriente alterna (o circuitos AC, del inglés "Alternating Current").

Llamamos corriente continua a aquella en la que los electrones circulan a través del conductor siempre en la misma dirección (es decir, en la que los extremos de mayor y menor potencial –o lo que es lo mismo, los polos positivo y negativo– son siempre los mismos). Aunque comúnmente se identifica la corriente continua con la corriente constante (por ejemplo, la suministrada por una batería), estrictamente solo es continua toda corriente que, tal como acabamos de decir, mantenga siempre la misma polaridad.

Llamamos corriente alterna a aquella en la que la magnitud y la polaridad del voltaje (y por tanto, las de la intensidad también) varían cíclicamente. Esto último implica que los polos positivo y negativo se intercambian alternativamente a lo largo del tiempo y, por tanto, que el voltaje va tomando valores positivos y negativos con una frecuencia determinada.

La corriente alterna es el tipo de corriente que llega a los hogares y empresas proveniente de la red eléctrica general. Esto es así porque la corriente alterna es más fácil y eficiente de transportar a lo largo de grandes distancias (ya que sufre menos pérdidas de energía) que la corriente continua. Además, la corriente alterna puede ser convertida a distintos valores de tensión (ya sea aumentándolos o disminuyéndolos según nos interese a través de un dispositivo llamado transformador) de una forma más sencilla y eficaz.

No obstante, en todos los proyectos de este libro utilizaremos tan solo corriente continua, ya que los circuitos donde podemos utilizar Arduino (y de hecho, la mayoría de circuitos electrónicos domésticos) solo funcionan correctamente con este tipo de corriente.

¿Qué es la resistencia eléctrica?

Podemos definir la resistencia eléctrica interna de un objeto cualquiera (aunque normalmente nos referiremos a algún componente electrónico que forme parte de nuestros circuitos) como su capacidad para oponerse al paso de la corriente eléctrica a través de él. Es decir, cuanto mayor sea la resistencia de ese componente,

más dificultad tendrán los electrones para atravesarlo, hasta incluso el extremo de imposibilitar la existencia de electricidad.

Esta característica depende entre otros factores del material con el que está construido ese objeto, por lo que podemos encontrarnos con materiales con poca o muy poca resistencia intrínseca (los llamados "conductores", como el cobre o la plata) y materiales con bastante o mucha resistencia (los llamados "aislantes", como la madera o determinados tipos de plástico, entre otros). No obstante, hay que insistir en que aunque un material sea conductor, siempre poseerá inevitablemente una resistencia propia que evita que se transfiera el 100% de la corriente a través de él, por lo que incluso un simple cable de cobre tiene cierta resistencia interna (normalmente despreciable, eso sí) que reduce el flujo de electrones original.

La unidad de medida de la resistencia de un objeto es el ohmio (Ω). También podemos hablar de kilohmios (1 kΩ = 1000 Ω), de megaohmios (1 MΩ = 1000 kΩ), etc.

¿Qué es la Ley de Ohm?

La Ley de Ohm dice que si un componente eléctrico con resistencia interna, R, es atravesado por una intensidad de corriente, I, entre ambos extremos de dicho componente existirá una diferencia de potencial, V, que puede ser conocida gracias a la relación **V = I·R**.

De esta fórmula es fácil deducir relaciones de proporcionalidad interesantes entre estas tres magnitudes eléctricas. Por ejemplo: se puede ver que (suponiendo que la resistencia interna del componente no cambia) cuanto mayor es la intensidad de corriente que lo atraviesa, mayor es la diferencia de potencial entre sus extremos. También se puede ver que (suponiendo en este caso que en todo momento circula la misma intensidad de corriente por el componente), cuanto mayor es su resistencia interna, mayor es la diferencia de potencial entre sus dos extremos.

Además, despejando la magnitud adecuada de la fórmula anterior, podemos obtener, a partir de dos datos conocidos cualesquiera, el tercero. Por ejemplo, si conocemos V y R, podremos encontrar I mediante **I = V/R**, y si conocemos V e I, podremos encontrar R mediante **R = V/I**.

A partir de las fórmulas anteriores debería ser fácil ver también por ejemplo que cuanto mayor es el voltaje aplicado entre los extremos de un componente (el cual suponemos que posee una resistencia de valor fijo), mayor es la intensidad de

corriente que pasa por él. O que cuanto mayor es la resistencia del componente (manteniendo constante la diferencia de potencial entre sus extremos), menor es la intensidad de corriente que pasa a través de él. De hecho, en este último caso, si el valor de la resistencia es suficientemente elevado, podemos conseguir incluso que el flujo de electrones se interrumpa.

¿Qué es la potencia?

Podemos definir la potencia de un componente eléctrico/electrónico como la energía consumida por este en un segundo. Si, no obstante, estamos hablando de una fuente de alimentación, con la palabra potencia nos referiremos entonces a la energía eléctrica aportada por esta al circuito en un segundo. En ambos casos (ya sea potencia consumida o generada), la potencia es un valor intrínseco propio del componente o generador, respectivamente. Su unidad de medida es el vatio (W), pero también podemos hablar de milivatios (1 mW = 0,001 W), o kilovatios (1 kW = 1000 W).

A partir de la potencia conocida propia del componente/generador y del tiempo que este esté funcionando, se puede conocer la energía consumida/aportada total, mediante la expresión: **E = P · t** .

Cuando una fuente de alimentación aporta una determinada energía eléctrica, esta puede ser consumida por los distintos componentes del circuito de diversas maneras: la mayoría de veces es gastada en forma de calor debido al efecto de las resistencias internas intrínsecas de cada componente (el llamado “efecto Joule”), pero también puede ser consumida en forma de luz (si ese componente es una bombilla, por ejemplo) o en forma de movimiento (si ese componente es un motor, por ejemplo), o en forma de sonido (si ese componente es un altavoz, por ejemplo), o en una mezcla de varias.

Podemos calcular la potencia consumida por un componente eléctrico si sabemos el voltaje al que está sometido y la intensidad de corriente que lo atraviesa, utilizando la fórmula **P = V·I**. Por ejemplo, una bombilla sometida a 220 V por la que circula 1 A consumirá 220 W. Por otro lado, a partir de la Ley de Ohm podemos deducir otras dos fórmulas equivalentes que nos pueden ser útiles si sabemos el valor de la resistencia R interna del componente: $P = I^2 \cdot R$ o también $P = V^2/R$.

Finalmente, hay que saber que los materiales conductores pueden soportar hasta una cantidad máxima de potencia consumida, más allá de la cual se corre el riesgo de sobrecalentarlos y dañarlos.

¿Qué son las señales digitales y las señales analógicas?

Podemos clasificar las señales eléctricas (ya sean voltajes o intensidades) de varias maneras según sus características físicas. Una de las clasificaciones posibles es distinguir entre señales digitales y señales analógicas.

Señal digital es aquella que solo tiene un número finito de valores posibles (lo que se suele llamar "tener valores discretos"). Por ejemplo, si consideramos como señal el color emitido por un semáforo, es fácil ver que esta es de tipo digital, porque solo puede tener tres valores concretos, diferenciados y sin posibilidad de transición progresiva entre ellos: rojo, ámbar y verde.

Un caso particular de señal digital es la señal binaria, donde el número de valores posibles solo es 2. Conocer este tipo de señales es importante porque en la electrónica es muy habitual trabajar con voltajes (o intensidades) con tan solo dos valores. En estos casos, uno de los valores del voltaje binario suele ser 0 –o un valor aproximado– para indicar precisamente la ausencia de voltaje, y el otro valor puede ser cualquiera, pero lo suficientemente distinguible del 0 como para indicar sin ambigüedades la presencia de señal. De esta forma, un valor del voltaje binario siempre identifica el estado "no pasa corriente" (también llamado estado "apagado" –"off" en inglés– , BAJO –LOW en inglés–, o "0") y el otro valor siempre identifica el estado "pasa corriente" (también llamado "encendido" –"on" – , ALTO –HIGH – , o "1").

El valor de voltaje concreto que se corresponda con el estado ALTO será diferente según los dispositivos electrónicos utilizados en cada momento. En los proyectos de este libro, por ejemplo, será habitual utilizar valores de 3,3 V o 5 V. Pero atención: es importante tener en cuenta que si sometemos un dispositivo electrónico a un voltaje demasiado elevado (por ejemplo, si aplicamos 5V como valor ALTO cuando el dispositivo solo admite 3,3 V) corremos el riesgo de dañarlo irreversiblemente.

Además de los niveles ALTO y BAJO, en una señal binaria existen las transiciones entre estos niveles (de ALTO a BAJO y de BAJO a ALTO), denominadas flanco de bajada y de subida, respectivamente.

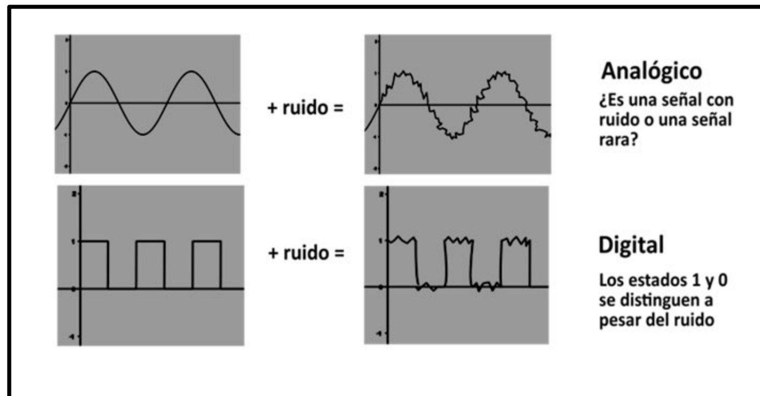
Señal analógica es aquella que tiene infinitos valores posibles dentro de un rango determinado (lo que se suele llamar "tener valores continuos"). La mayoría de

magnitudes físicas (temperatura, sonido, luz...) son analógicas, así como también las más específicamente eléctricas (voltaje, intensidad, potencia…) porque todas ellas, de forma natural, pueden sufrir variaciones continuas sin saltos.

No obstante, muchos sistemas electrónicos (un computador, por ejemplo) no tienen la capacidad de trabajar con señales analógicas: solamente pueden manejar señales digitales (especialmente de tipo binario; de ahí su gran importancia). Por tanto, necesitan disponer de un conversor analógico-digital que "traduzca" (mejor dicho, "simule") las señales analógicas del mundo exterior en señales digitales entendibles por dicho sistema electrónico. También se necesitará un conversor digital-analógico si se desea realizar el proceso inverso: transformar una señal digital interna del computador en una señal analógica para poderla así emitir al mundo físico. Un ejemplo del primer caso sería la grabación de un sonido mediante un micrófono, y uno del segundo caso sería la reproducción de un sonido pregrabado mediante un altavoz.

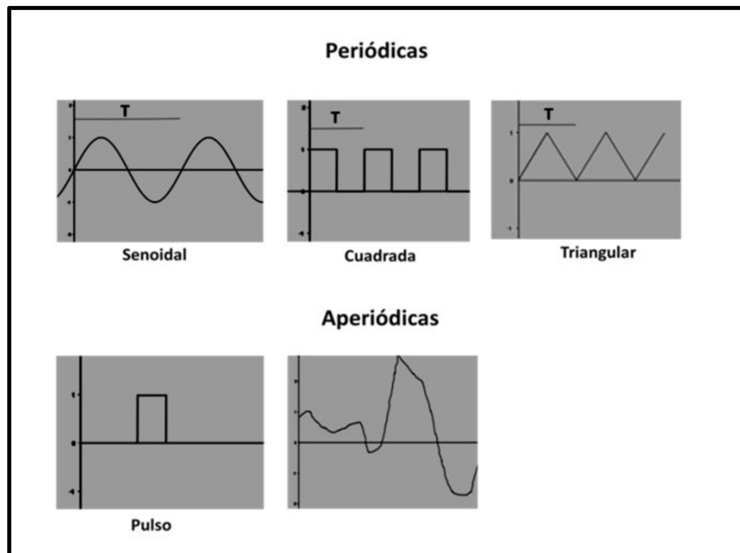
Sobre los métodos utilizados para realizar estas conversiones de señal analógica a digital, y viceversa, ya hablaremos extensamente más adelante, pero lo que debemos saber ya es que, sea cual sea el método utilizado, siempre existirá una pérdida de información (de "calidad") durante el proceso de conversión de la señal. Esta pérdida aparece porque es matemáticamente imposible realizar una transformación perfecta de un número infinito de valores (señal analógica) a un número finito (señal digital) debido a que, por fuerza, varios valores de la señal analógica deben "colapsar" en un único valor indistinguible de la señal digital.

A pesar de lo anterior, la razón por la cual la mayoría de sistemas electrónicos utilizan para funcionar señales digitales en vez de analógicas es porque las primeras tienen una gran ventaja respecto las segundas: son más inmunes al ruido. Por "ruido" se entiende cualquier variación no deseada de la señal, y es un fenómeno que ocurre constantemente debido a una gran multitud de factores. El ruido modifica la información que aporta una señal y afecta en gran medida al correcto funcionamiento y rendimiento de los dispositivos electrónicos. Si la señal es analógica, el ruido es mucho más difícil de tratar y la recuperación de la información original se complica.



¿Qué son las señales periódicas y las señales aperiódicas?

Otra clasificación que podemos hacer con las señales eléctricas es dividirlas entre señales periódicas y aperiódicas. Llamamos señal periódica a aquella que se repite tras un cierto período de tiempo (T) y señal aperiódica a aquella que no se repite. En el caso de las primeras (las más interesantes con diferencia), dependiendo de cómo varíe la señal a lo largo del tiempo, esta puede tener una “forma” concreta (senoidal –es decir, que sigue el dibujo de la función seno–, cuadrada, triangular, etc.).



Las señales periódicas tienen una serie de características que debemos identificar y definir para poder trabajar con ellas de una forma sencilla:

Frecuencia (f): es el número de veces que la señal se repite en un segundo. Se mide en hercios (Hz), o sus múltiplos (como kilohercios o megahercios). Por ejemplo, si decimos que una señal es de diez hercios, significa que se repite diez veces cada segundo.

Período (T): es el tiempo que dura un ciclo completo de la señal, antes de repetirse otra vez. Es el inverso de la frecuencia (T = 1/f) y se mide en segundos.

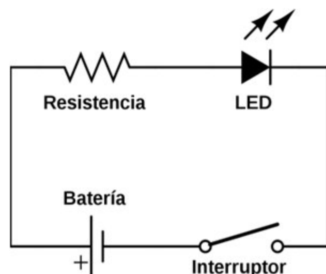
Valor instantáneo: es el valor concreto que toma la señal (voltaje, intensidad, etc.) en cada instante

Valor medio: es un valor calculado matemáticamente realizando la media de los diferentes valores que ha ido teniendo la señal a lo largo de un tiempo concreto. Algunos componentes electrónicos (por ejemplo, algunos motores) responden no al valor instantáneo sino al valor medio de la señal.

CIRCUITOS ELÉCTRICOS BÁSICOS

Representación gráfica de circuitos

Para describir de una forma sencilla y clara la estructura y la composición de un circuito eléctrico se utilizan esquemas gráficos. En ellos se representa cada dispositivo del circuito mediante un símbolo estandarizado y se dibujan todas las interconexiones existentes entre ellos. Por ejemplo, un circuito muy simple sería:



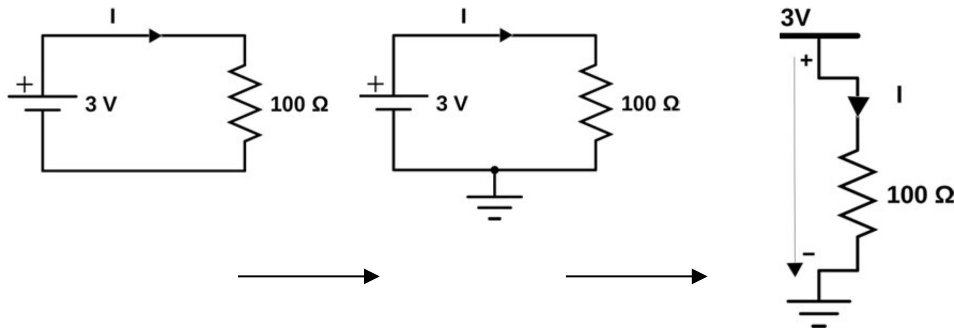
En el esquema anterior podemos apreciar cuatro dispositivos (presentes prácticamente en cualquier circuito) representados por su símbolo convencional: una

pila o batería (cuya tarea es alimentar eléctricamente al resto de componentes), una resistencia (componente específicamente diseñado para oponerse al paso de la corriente, de ahí su nombre), un LED (componente que se ilumina cuando recibe corriente) y un interruptor. En este ejemplo, la batería creará la diferencia de potencial necesaria entre sus dos extremos –también llamados "bornes" o "polos"– para que se genere una corriente eléctrica, la cual surgirá desde su polo positivo (el marcado con el signo "+"), pasará a través de la resistencia, pasará seguidamente a través del LED (iluminándolo, por tanto) y llegará a su destino final (el polo negativo de la batería) siempre y cuando el interruptor cierre el circuito.

Aclaremos lo que significa "cerrar un circuito". Acabamos de decir que si existe una diferencia de potencial, aparecerá una corriente eléctrica que siempre circula desde el polo positivo de la pila hasta el negativo. Pero esto solo es posible si existe entre ambos polos un camino (el circuito propiamente dicho) que permita el paso de dicha corriente. Si el circuito está abierto, a pesar de que la batería esté funcionando, la corriente no fluirá. La función de los interruptores es precisamente cerrar o abrir el circuito para que pueda pasar la corriente o no, respectivamente. En el esquema siguiente esto se ve más claro:



Por otro lado, los circuitos se pueden representar alternativamente de una forma ligeramente diferente a la mostrada anteriormente, utilizando para ello el concepto de "tierra" (también llamado "masa"). La "tierra" ("ground" en inglés) es simplemente un punto del circuito que elegimos arbitrariamente como referencia para medir la diferencia de potencial existente entre este y cualquier otro punto del circuito. En otras palabras: el punto donde diremos que el voltaje es 0. Por utilidad práctica, normalmente el punto de tierra se asocia al polo negativo de la pila. Este nuevo concepto nos simplificará muchas veces el dibujo de nuestros circuitos, ya que si representamos el punto de tierra con el símbolo ⏚, los circuitos se podrán dibujar de la siguiente manera:



También podremos encontrarnos con esquemas eléctricos que muestren intersecciones de cables. En este caso, deberemos fijarnos si aparece dibujado un círculo en el punto central de la intersección. Si es así, se nos estará indicando que los cables están física y eléctricamente conectados entre sí. Si no aparece dibujado ningún círculo en el punto central de la intersección, se nos estará indicando que los cables son vías independientes que simplemente se cruzan en el espacio.

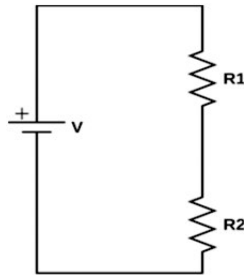
Conexiones en serie y en paralelo

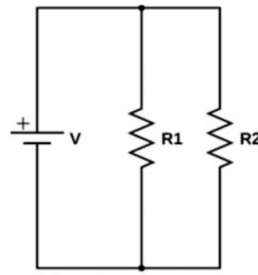
Los distintos dispositivos presentes en un circuito pueden conectarse entre sí de varias formas. Las más básicas son la "conexión en serie" y la "conexión en paralelo". De hecho, cualquier otro tipo de conexión, por compleja que sea, es una combinación de alguna de estas dos.

Si diversos componentes se conectan entre sí en paralelo, a todos ellos se les aplica la misma tensión por igual (es decir, cada componente trabaja al mismo voltaje). Por otro lado, la intensidad de corriente total será la suma de las intensidades que pasan por cada componente, ya que existen varios caminos posibles para el paso de los electrones.

Si la conexión es en serie, la tensión total disponible se repartirá (normalmente, de forma desigual) entre los diferentes componentes, de manera que cada uno trabaje sometido a una parte de la tensión total. Es decir: la tensión total será la suma de las tensiones en cada componente. Por otro lado, la intensidad de corriente que circulará por todos los componentes en serie será siempre la misma, ya que solo existe un camino posible para el paso de los electrones.

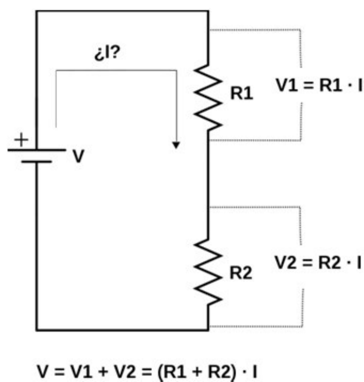
Se puede entender mejor la diferencia mediante los siguientes esquemas, en los que se puede ver la conexión en serie y en paralelo de dos resistencias.





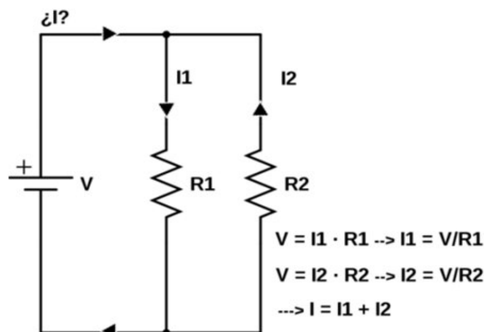
Gracias a la Ley de Ohm podemos obtener el valor de alguna magnitud eléctrica (V, I o R) si conocemos previamente el valor de alguna otra involucrada en el mismo circuito. Para ello, debemos tener en cuenta las particularidades de las conexiones en serie o en paralelo.

Veamos esto usando como ejemplo el circuito de las dos resistencias en serie:



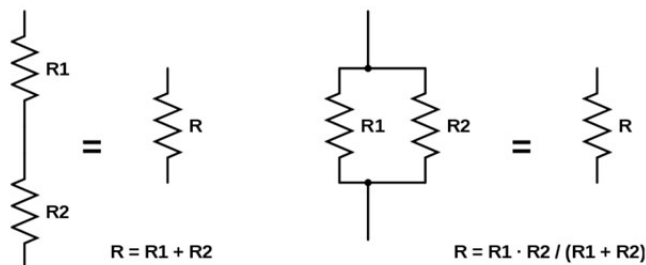
En el esquema anterior V1 representa el voltaje aplicado a R1 y V2 el voltaje aplicado a R2. Si tenemos por ejemplo una fuente de alimentación eléctrica (una pila) que aporta un voltaje de 10 V y dos resistencias cuyos valores son R1 = 1Ω y R2 = 4Ω respectivamente, para calcular la intensidad que circula tanto por R1 como por R2 (recordemos que es la misma porque solo existe un único camino posible) simplemente deberíamos realizar la siguiente operación: I = 10 V/(1 Ω + 4 Ω) = 2 A , tal como se muestra en el esquema anterior.

Veamos ahora el circuito de las dos resistencias en paralelo:



En el esquema anterior I1 representa la intensidad de corriente que atraviesa R1 e I2 la intensidad de corriente que atraviesa R2. Si tenemos por ejemplo una fuente de alimentación eléctrica (una pila) que aporta un voltaje de 10 V y dos resistencias cuyos valores son R1 = 1Ω y R2=4Ω, respectivamente, para calcular la intensidad que circula por R1 deberíamos realizar (tal como se muestra en el esquema) la siguiente operación: I1 = 10 V/1 Ω = 10A ; para calcular la intensidad que circula por R2 deberíamos hacer: I2 = 10 V/4 Ω = 2,5A ; y la intensidad total que circula por el circuito sería la suma de las dos: I = I1 + I2 = 10 A + 2,5 A = 12,5 A.

A partir de los ejemplos anteriores, podemos deducir un par de fórmulas que nos vendrán bien a lo largo de todo el libro para simplificar los circuitos. Si tenemos dos resistencias conectadas en serie o en paralelo, es posible sustituirlas en nuestros cálculos por una sola resistencia cuyo comportamiento sea totalmente equivalente. En el caso de la conexión en serie, el valor de dicha resistencia (R) vendría dado por **R = R1+R2**, y en el caso de la conexión en paralelo, su valor equivalente se calcularía mediante la fórmula **R = (R1 · R2)/(R1 + R2)**, tal como se puede ver en el siguiente diagrama.

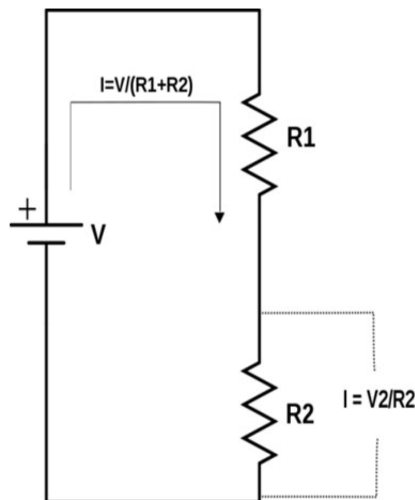


Un dato interesante de tener en cuenta (que se deduce de la propia fórmula) es que cuando se conectan resistencias en paralelo, el valor de R resultante siempre es menor que el menor valor de las resistencias implicadas.

El divisor de tensión

El "divisor de tensión" no es más que un circuito formado por una resistencia conectada en serie con cualquier otro dispositivo eléctrico. Su intención es reducir la tensión aplicada a dicho dispositivo, estableciéndola en un valor seguro para no dañarlo. Dicho de otra forma: el "divisor de tensión" sirve para obtener un voltaje menor que un cierto voltaje original.

La mayor o menor cantidad de reducción que consigamos en la tensión final dependerá del valor de la resistencia que utilicemos como divisor: a mayor valor de resistencia, mayor reducción. De todas formas, hay que tener en cuenta además que la tensión obtenida asimismo depende del valor de la tensión original: si aumentamos esta, aumentaremos proporcionalmente aquella también. Todos estos valores los podemos calcular fácilmente usando un ejemplo concreto, como el del esquema siguiente.



Tal como se puede ver, tenemos una fuente de alimentación eléctrica (una pila) que aporta un voltaje de 10 V y dos resistencias cuyos valores son R1 = 1Ω (la cual hará de divisor de tensión) y R2 = 4 Ω, respectivamente. Sabemos además que la intensidad I es siempre la misma en todos los puntos del circuito –ya que no hay ramificaciones en paralelo–. Por lo tanto, para calcular V2 (es decir, el voltaje aplicado a R2, el cual ha sido rebajado respecto al aportado por la pila gracias a R1), nos podemos dar cuenta de que I = V2/R2 y que I = V/(R1 + R2) , por lo que de aquí es fácil obtener que V2 = (R2 · V)/(R1 + R2). Queda entonces claro de la expresión

anterior lo dicho en el párrafo anterior: que V2 siempre será proporcionalmente menor a V, y según sea R1 mayor, V2 será menor.

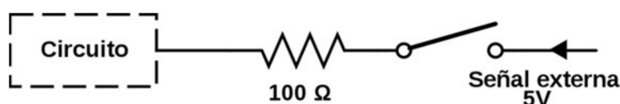
Las resistencias “pull-up” y “pull-down”

Muchas veces, los circuitos eléctricos tienen “entradas” por las que reciben una señal eléctrica del exterior (de tipo binario) que no tiene nada que ver con la señal de alimentación obtenida de la fuente. Estas señales externas pueden servir para multitud de cosas: para activar o desactivar partes del circuito, para enviar al circuito información de su entorno, etc.

Las resistencias “pull-up” (y “pull-down”) son resistencias normales, solo que llevan ese nombre por la función que cumplen: sirven para asumir un valor por defecto de la señal recibida en una entrada del circuito cuando por ella no se detecta ningún valor concreto (ni ALTO ni BAJO), que es lo que ocurre cuando la entrada no está conectada a nada (es decir, está “al aire”). Así pues, este tipo de resistencias aseguran que los valores binarios recibidos no fluctúan sin sentido en ausencia de señal de entrada.

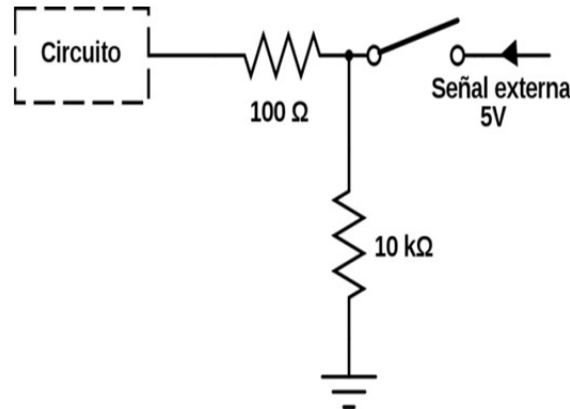
En las resistencias “pull-up” el valor que se asume por defecto cuando no hay ningún dispositivo externo emisor de señal conectado a la entrada es ALTO y en las “pull-down” es el valor BAJO, pero ambas persiguen el mismo objetivo, así que la elección de una resistencia de tipo “pull-up” o “pull-down” dependerá de las circunstancias particulares de nuestro montaje. La diferencia entre unas y otras está en su ubicación dentro del circuito: las resistencias “pull-up” se conectan directamente a la fuente de señal externa y las “pull-down” directamente a tierra (ver diagramas siguientes).

Veamos un ejemplo concreto de la utilidad de una resistencia “pull-down”. Supongamos que tenemos un circuito como el siguiente (donde la resistencia de 100 ohmios no es más que un divisor de tensión colocado en la entrada del circuito para protegerla).



Cuando el interruptor esté pulsado, la entrada del circuito estará conectada a una señal de entrada válida, que supondremos binaria (es decir, que tendrá dos posibles valores: ALTO –de 5V, por ejemplo– y BAJO –de 0V–), por lo que el circuito recibirá alguno de estos dos valores concretos y todo estará ok. En cambio, si el

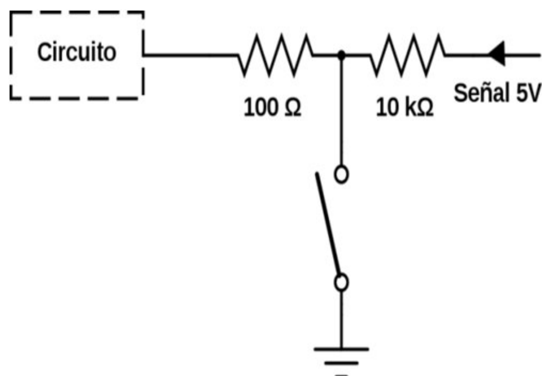
interruptor se deja de pulsar, el circuito se abrirá y la entrada del circuito no estará conectada a nada. Esto implica que habrá una señal de entrada fluctuante (también llamada “flotante” o “inestable”) que no nos interesa. La solución en este caso sería colocar una resistencia “pull-down” así:



En este ejemplo la resistencia “pull-down” es de 10 KΩ. Cuando el interruptor esté pulsado, la entrada del circuito estará conectada a una señal de entrada válida, como antes. Cuando el interruptor se deje de pulsar, la entrada del circuito estará conectada a la resistencia “pull-down”, la cual tira hacia tierra (que es una referencia siempre fija).

Alguien podría pensar que cuando el interruptor esté pulsado, el circuito recibirá la señal de entrada pero también estará conectado a tierra a través de la resistencia “pull-down”: ¿qué pasa realmente entonces? Aquí está la clave de por qué se usa la resistencia “pull-down” y no se usa una conexión directa a tierra: la oposición al paso de los electrones provenientes de la señal externa que ejerce la resistencia “pull-down” provoca que estos se desvíen siempre a la entrada del circuito. Si hubiéramos conectado la entrada del circuito a tierra directamente sin usar la resistencia “pull-down”, la señal externa se dirigiría directamente a tierra sin pasar por la entrada del circuito porque por ese camino encontraría menor resistencia (pura Ley de Ohm: menos resistencia, más intensidad).

Con una resistencia “pull-up” se podría haber conseguido lo mismo, tal como muestra el siguiente esquema. En este caso, cuando el interruptor está pulsado la señal exterior se desvía a tierra porque encuentra un camino directo a ella (por lo que la entrada del circuito no recibe nada –un “0”–) y cuando el interruptor se deja sin pulsar es cuando la entrada del circuito recibe la señal exterior. Hay que tener cuidado con esto.



En los ejemplos anteriores hemos utilizado resistencias “pull-up” o “pull-down” de 10 KΩ. Es una norma bastante habitual utilizar este valor concreto en proyectos de electrónica donde se trabaja en el rango de los 5V, aunque, en todo caso, si queremos afinarlo algo más, podemos calcular su valor ideal utilizando la Ley de Ohm a partir de la corriente que consuma el circuito.

FUENTES DE ALIMENTACIÓN ELÉCTRICA

Tipos de pilas/baterías

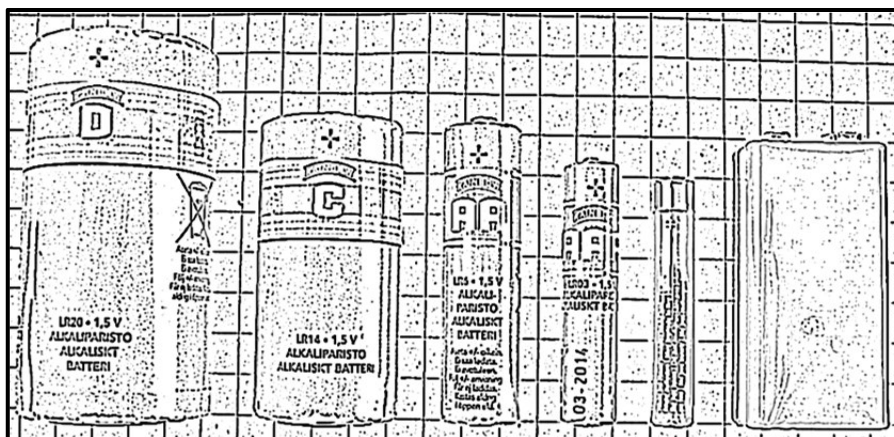
Llamamos fuente de alimentación eléctrica al elemento responsable de generar la diferencia de potencial necesaria para que fluya la corriente eléctrica por un circuito y así puedan funcionar los dispositivos conectados a este. Las fuentes que utilizaremos más a menudo en nuestros proyectos serán de dos tipos: las pilas o baterías y los adaptadores AC/DC.

El término “pila” sirve para denominar a los generadores de electricidad basados en procesos químicos normalmente no reversibles y, por tanto, son generadores no recargables; mientras que el término “batería” se aplica generalmente a dispositivos electroquímicos semi-reversibles que permiten ser recargados, aunque estos términos no son una definición formal estricta. El término “acumulador” se aplica indistintamente a uno u otro tipo (así como a otros tipos de generadores de tensión, como los condensadores eléctricos) siendo pues un término neutro capaz de englobar y describir a todos ellos.

Si distinguimos las pilas/baterías por la disolución química interna responsable de la generación de la diferencia de potencial entre sus polos,

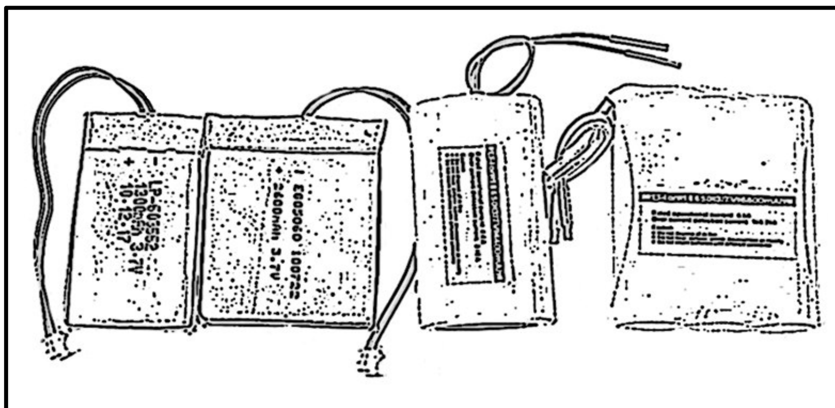
encontraremos que las pilas ("acumuladores no recargables") más extendidas actualmente en el mercado son las de tipo alcalino, y las baterías ("acumuladores recargables") más habituales son por un lado las de níquel-cadmio (Ni-Cd) y sobre todo níquel-hidruro metálico (NiMH), y por otro las de ion-litio (Li-ion) y las de polímero de ion-litio (LiPo). De todos estos tipos de baterías, las LiPo son las que tienen una densidad de carga más elevada (es decir, que siendo las más ligeras son las que tienen, no obstante, más autonomía) pero son más caras.

La industria internacional sigue unas normas comunes de estandarización para la fabricación de pilas de tipo alcalino y baterías de tipo Ni-Cd/NiMH que definen unos determinados tamaños, formas y voltajes preestablecidos, de manera que se puedan utilizar sin problemas en cualquier aparato eléctrico a nivel mundial. En este sentido, los tipos de pilas más habituales son las de tipo D (LR20), C (LR14), AA (LR06) y AAA (LR03), todas ellas generadoras de 1,5 V y de forma cilíndrica aunque de dimensiones diferentes (de hecho, se han listado de mayor tamaño a menor). También son frecuentes las de tipo PP3 (6LR61), que generan 9 V y tienen forma de prisma rectangular; y las de tipo 3R12 (de "petaca") que generan 4,5 V y tienen forma cilíndrica achatada. En la imagen siguiente se pueden apreciar, de izquierda a derecha, acumuladores –alcalinos– de tipo D, C, AA, AAA, AAAA y PP3, colocados sobre un papel cuadriculado.



En la imagen siguiente, a la izquierda se muestran dos baterías de tipo LiPo y a la derecha dos encapsulados hechos de baterías cilíndricas de tipo Li-ion. Las primeras suelen venir en forma de delgados rectángulos dentro de una bolsa plateada y las segundas suelen venir dentro de una carcasa dura rectangular o cilíndrica, aunque ambas vienen realmente en una gran versatilidad y flexibilidad de formas y tamaños. Las LiPo son más ligeras que las Li-ion pero suelen tener una

capacidad menor, por eso las primeras se suelen utilizar en aparatos pequeños como teléfonos móviles y las segundas en cargadores de portátiles y similares.



También hemos de indicar la existencia de las pilas de tipo “botón”. Hay de muchos tipos: si están fabricadas con litio-dióxido de manganeso, su nomenclatura empieza con “CR” (así, podemos tener la CR2032, la CR2477, etc.) y, aunque cada una de ellas tenga un encapsulado con diámetro y anchura diferente, todas generan 3 V. Si están fabricadas con óxido de plata, su nomenclatura comúnmente empieza con “SR” o “SG” (así, podemos tener la SR44, la SR58, etc., dependiendo de sus dimensiones). También existen de tipo alcalinas, cuyo código comúnmente empieza por “LR” o “AG”. Tanto las de óxido de plata como las alcalinas generan 1,5 V. En cualquier caso, sea del tipo que sea, en todas las pilas botón el terminal negativo es la tapa y el terminal positivo es el metal de la otra cara.

Características de las pilas/baterías

Hay que tener en cuenta que el voltaje que aportan las distintas pilas es un valor “nominal”: es decir, por ejemplo una pila AA de 1,5 V en realidad al principio de su vida útil genera unos 1,6 V, rápidamente desciende a 1,5 V y entonces poco a poco va descendiendo hasta 1 V, momento en el cual la pila la podemos considerar “gastada”. Lo mismo pasa con los otros tipos de batería; por ejemplo, una batería LiPo marcada como “3,7 V/(4,2 V)” indica que inicialmente es capaz de aportar un voltaje máximo de 4,2 V pero rápidamente desciende a 3,7 V, el cual será su voltaje medio durante la mayor parte de su vida útil, hasta que finalmente baje rápidamente hasta los 3 V y automáticamente deje de funcionar. En este sentido, es útil consultar la documentación oficial ofrecida por el fabricante para cada batería particular (el llamado “datasheet” de la batería) para saber la variación del voltaje aportado en función del tiempo de funcionamiento.

Además del voltaje generado de una pila/batería (que asumiremos a partir de ahora siempre constante) hay que conocer otra característica importante: la carga eléctrica que esta es capaz de almacenar (a veces llamada "capacidad" de la pila/batería). Este valor se mide en amperios-hora (Ah), o miliamperios-hora (mAh) y nos permite saber aproximadamente cuánta intensidad de corriente puede aportar la pila/batería durante un determinado período de tiempo. En este sentido, hay que recordar que mientras el voltaje aportado por la pila/batería es idealmente constante, la intensidad aportada, en cambio, varía en cada momento según lo haga el consumo eléctrico del circuito al que la conectemos. Por ejemplo, 1 Ah significa que en teoría la pila/batería puede ofrecer durante una hora una intensidad de 1 A (si así lo requiere el circuito), o 0,1 A durante 10 horas, o 0,01 A durante 100 horas, etc., pero siempre al mismo voltaje.

Sin embargo, lo anterior no es exactamente así, porque cuanta más cantidad de corriente la pila/batería aporte, en realidad su tiempo de funcionamiento se reducirá en una proporción mucho mayor a la marcada por su capacidad. Por ejemplo, una pila botón de 1 Ah es incapaz de aportar 1 A durante una hora entera (ni tan siquiera 0,1 A en 10 horas) porque se agota mucho antes, pero en cambio, no tiene problemas en aportar 0,001 A durante 1000 horas. Para saber la intensidad de corriente concreta que hace cumplir el valor nominal de Ah de una batería, deberemos consultar la documentación del fabricante (el "datasheet" de la batería). Esta intensidad de corriente "óptima" en el caso de las baterías LiPo se suele llamar "capabilidad", y viene expresada en unidades C, donde una unidad C se corresponde con el valor de Ah de esa batería dividido por una hora.

Por ejemplo, la unidad C de una batería con carga de 2 Ah será de 2 A, y su capabilidad concreta será una cantidad determinada de unidades C, consultable en el datasheet (1C, 2C…). Si tenemos entonces, por ejemplo, una batería de 2 Ah y 0,5 C y otra de 2 Ah y 2C, la primera podrá aportar una corriente estable de hasta 1 A sin agotarse prematuramente, y la segunda podrá aportar una corriente de hasta 4 A. Sabido esto, hay que tener en cuenta por ejemplo que las pilas botón tienen una capabilidad muy pequeña (0,01C es un valor habitual), por lo que si son forzadas a aportar mucha intensidad en un momento dado, su vida se reducirá drásticamente.

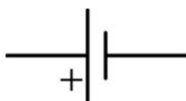
Por otro lado, ya hemos dicho que el que una pila aporte en un determinado momento una intensidad de corriente u otra depende básicamente del consumo eléctrico (medido en amperios, o más normalmente en miliamperios) que realice el conjunto total de dispositivos que estén conectados en ese momento al circuito (y que lógicamente puede ser muy variado según el caso). Es decir: el tiempo de funcionamiento de una batería depende de la demanda del circuito al que está

conectado. De forma más concreta, podemos obtener (de forma muy aproximada) el tiempo de descarga de una pila/batería mediante la expresión: *tiempo de descarga = capacidad batería / consumo eléctrico circuito.*

Por ejemplo, si una batería posee una carga eléctrica de 1000 mAh y un dispositivo consume 20 mA, la batería tardará 50 horas en descargarse; si en cambio el dispositivo consume 100 mA, esa batería tardará solamente 10 horas en descargarse. Todo esto es la teoría, ya que el valor numérico de mAh impreso sobre la batería debe ser tomado solo como una aproximación, y debe tenerse en cuenta solamente en los rangos niveles de consumo (medidos en unidades C) especificados por el fabricante, ya que para altos consumos ya sabemos que este valor no puede ser extrapolado con precisión.

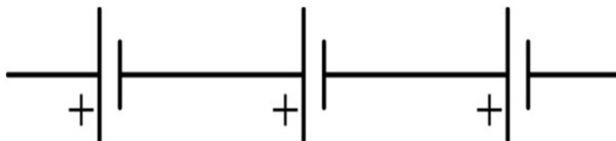
Conexiones de varias pilas/baterías

Ya hemos visto en diagramas anteriores que el símbolo que se suele utilizar en el diseño de circuitos electrónicos para representar una pila o batería es:



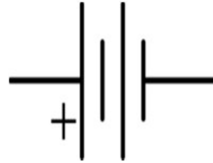
donde la parte más larga (y a veces pintada más gruesa) del dibujo representa el polo positivo de la fuente. A menudo se omite el símbolo "+".

Cuando hablamos de conectar pilas "en serie" queremos decir que conectamos el polo negativo de una con el polo positivo de otra, y así, de tal forma que finalmente tengamos un polo positivo global por un lado y un polo negativo global por otro. En esta figura se puede entender mejor:



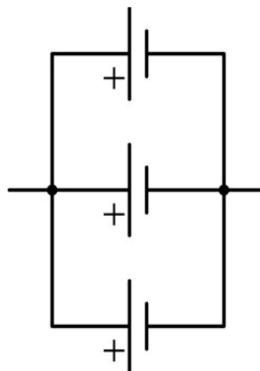
La conexión en serie de baterías es útil cuando necesitamos tener una batería que genere un determinado voltaje relativamente elevado (por ejemplo, 12 V) y solo disponemos de pilas de menor voltaje (por ejemplo, de 1,5 V). En ese caso, podemos conectar dichas unidades de 1,5 V en serie para obtener "a mano" la batería que aporte el voltaje deseado, ya que el voltaje total aportado por baterías conectadas en serie es la suma de sus voltajes individuales. En nuestro ejemplo, para obtener 12 V a

partir de pilas de 1,5 V, necesitaríamos 8 unidades, porque 1,5 V · 8 = 12 V. De hecho, las pilas comerciales de 4,5 V y 9 V (y de 6 V y 12 V, que también las hay) suelen fabricarse conectando internamente en serie pilas de 1,5 V. Por eso, muchas veces veremos el siguiente símbolo (en vez del anteriormente mostrado) representando una pila:



No obstante, también hay que tener en cuenta que la capacidad total (es decir, los mAh del conjunto de pilas en serie) no aumenta: seguirá siendo exactamente la misma que la que tenga una batería de ese conjunto de forma individual e independiente. Este hecho es importante porque generalmente los circuitos que necesitan ser alimentados con grandes voltajes tiene un mayor consumo eléctrico, por lo que (en virtud de la fórmula del apartado anterior) el tiempo de funcionamiento de una fuente formada por pilas en serie será bastante reducido.

Otra manera de conectar entre sí diferentes pilas individuales es en paralelo: en esta configuración, todos los polos del mismo signo están unidos entre sí. Es decir: por un lado se conectan los polos negativos de cada pila y por otro lado se conectan todos los polos positivos, siendo estos dos puntos comunes de unión los polos negativo y positivo globales.



Un conjunto de pilas en paralelo ofrece el mismo voltaje que una sola pila individual (es decir, si tenemos por ejemplo cuatro pilas de 1,5 V conectadas en paralelo, este conjunto dará igualmente un voltaje total de 1,5 V). La ventaja que

logramos es que la duración del sistema manteniendo esa tensión es mayor que si usamos una pila única, debido a que la capacidad (los mAh) del conjunto es la suma total de las capacidades de cada una de las pilas individuales.

Es muy importante asegurarse de que las pilas/baterías conectadas en serie o en paralelo sean del mismo tipo (alcalinas, NiMH, etc.), sean de la misma forma (AA, PP3, etc.) y aporten el mismo voltaje. Si no se hace así, el funcionamiento del conjunto puede ser inestable e incluso peligroso: en el caso de las baterías LiPo, pueden llegar hasta explotar si no se sigue esta norma. De hecho, en este tipo de baterías se recomienda adquirir packs (en serie o en paralelo) preensamblados, ya que nos ofrecen la garantía de que sus unidades han sido seleccionadas para tener la misma capacidad, resistencia interna, etc., y no causar problemas.

Compra de pilas/baterías

Cualquier tipo de pila/batería que necesitemos en nuestros proyectos (de diferente voltaje, capacidad, composición química...) lo podremos adquirir a través de cualquiera de los distribuidores listados en el apéndice A del libro: solo hay que utilizar el buscador que ofrecen en sus tiendas online para encontrar el tipo de pila/batería deseado. Por ejemplo, si buscamos baterías LiPo, el distribuidor de componentes electrónicos Sparkfun ofrece, entre otras, el producto nº 341 (a 3,7 V/850 mAh) o el nº 339 (a 3,7V/1Ah), ambas con conector JST de 2 pines. Si buscamos pilas alcalinas o recargables de tipo NiMH, es incluso más sencillo: los modelos más habituales están disponibles en cualquier comercio local.

Si precisamente se van a emplear este tipo de pilas (alcalinas o recargables NiMH) , lo más habitual es utilizar en nuestros proyectos algún tipo de portapilas que permita utilizar varias unidades conectadas en serie. Existen muchos modelos; como muestra, tan solo en la página web del distribuidor de componentes electrónicos Adafruit podemos encontrar (escribiendo su número de producto en el buscador integrado del portal), los siguientes productos:

Nº 248: portapilas de 6 unidades AA con clavija de 5,5/2,1 mm.
Nº 875: “ de 8 unidades AA con clavija de 5,5/2,1 mm e interruptor.
Nº 727: “ de 3 unidades AAA con conector tipo JST de 2 pines e interruptor.
Nº 67: “ de 1 unidad PP3 con clavija de 5,5/2,1mm e interruptor.
Nº 80: clip para conectar una 1 unidad PP3 con clavija de 5,5/2,1 mm
Nº 449: portapilas de 8 unidades AA con los cables directos (sin conector). Si se desea, los cables se pueden acoplar fácilmente a una clavija de 2,1 mm adquirida aparte (producto nº 369).

La clavija de 2,1 mm nos puede venir bien para alimentar nuestra placa Arduino ya que esta tiene un conector hembra de ese tipo. Por otro lado, el producto nº 727 está pensado para alimentar con pilas AAA circuitos originalmente diseñados para ser alimentados con baterías LiPo, ya que estas suelen incorporar precisamente un conector de tipo JST de 2 pines.

Compra de cargadores

Otro complemento que nos puede venir bien es un cargador de baterías. Por ejemplo, para las de tipo NiMH nos será suficiente con el ofrecido por ejemplo por Sparkfun con código de producto nº 10052. Este cargador se conecta directamente a un enchufe de pared y puede ser utilizado con baterías AA, AAA o PP3.

En cambio, si queremos recargar baterías LiPo, no vale cualquier cargador, hay que tener la precaución de usar siempre uno que cumpla dos condiciones: que aporte un voltaje igual (lo preferible) o menor que el voltaje máximo proporcionado por esa batería concreta, y que además aporte una intensidad igual o menor (lo preferible) que la capabilidad de esa batería concreta.

Si no se cumplen las dos condiciones anteriores, el cargador podría dañar la batería irreversiblemente (e incluso hacerla explotar). Las baterías LiPo son muy delicadas: también se corre el riesgo de explosión cuando se descargan por debajo de un voltaje mínimo (normalmente 3 V), o cuando son obligadas a aportar más corriente de la que pueden ofrecer (normalmente 2C), o cuando son utilizadas en ambientes de temperaturas extremas (normalmente fuera del rango 0 º-50 ºC), entre otras causas. Por eso muchas de estas baterías (aunque no todas) incorporan un circuito protector que detecta estas situaciones y desconecta la batería de forma segura. De todas formas, para conocer las características específicas de cada batería (como las tensiones, intensidades y temperaturas seguras) es obligatorio consultar la información que ofrece el fabricante para esa batería en concreto (su datasheet).

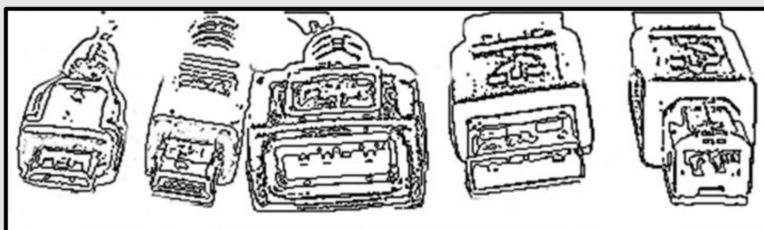
Los cargadores LiPo (y Li-ion) los podemos encontrar en multitud de establecimientos y en múltiples formas. Como ejemplos, tan solo en la página web de Sparkfun podemos encontrar los productos nº 10217, 10401, 11231 o 10161 (entre otros). Todos ellos son básicamente pequeñas plaquitas que disponen por un lado de un zócalo de tipo USB (mini-B o micro-B, según el caso) donde se conecta la alimentación externa y por otro lado de un zócalo JST de 2 pines, donde se conecta la batería LiPo a cargar. Todos ellos están preparados para ser alimentados con una fuente externa de 5 V; este voltaje lo puede ofrecer un cargador USB de pared de cualquier móvil actual o incluso un zócalo USB de nuestro propio computador. Gracias

al chip controlador MCP73831 que estas plaquitas llevan incorporado este voltaje de 5 V es rebajado convenientemente a los 3,7 V estándares que necesitan las pilas LiPo. No obstante, algunos de estos cargadores aportan a la batería una intensidad de 500 mA, otros de 1 A, etc., por lo que hay que conocer la capabilidad de nuestra batería para saber qué modelo de cargador es el más adecuado.

Cargadores similares a los descritos en el párrafo anterior los podemos encontrar en los otros distribuidores listados en el apéndice A y muchos más. Por ejemplo, en Adafruit ofrecen su producto nº 259, que es un cargador USB/JST como los anteriores y el producto nº 280, que además del zócalo USB incorpora una clavija hembra de 2,1 mm, permitiendo así obtener también la alimentación eléctrica de una fuente externa operativa entre 5 V y 12 V. En este sentido, el producto nº 8293 de Sparkfun es similar. Otra plaquita que incluso permite cargar baterías LiPo a partir de un panel solar conectable vía JST es el llamado “Lipo Rider” de Seeedstudio.

Breve nota sobre los conectores USB:

Cuando estamos hablando de “conector USB”, tenemos que tener claro de qué tipo de conector estamos hablando, porque existen varios modelos. En la imagen siguiente se muestran algunos de los más extendidos; de izquierda a derecha: conector micro-B macho, conector mini-B macho, conector A hembra, conector A macho y conector B macho.



Características de los adaptadores AC/DC

El otro tipo de fuente de alimentación externa, diferente de las pilas/baterías, que más utilizaremos para nuestros circuitos es el adaptador AC/DC. Su función típica es conectarse a una toma de la red eléctrica general para transformar el elevado voltaje alterno ofrecido por ella (en España es de 230 V $\pm$ 5% y 50 Hz $\pm$0,3%; si se desea saber el de otros países, se puede consultar http://kropla.com/electric2.htm)

en un voltaje continuo, constante y mucho menor, para ofrecer entonces este a los aparatos que se le conecten y así ponerlos en funcionamiento de una forma estable y segura.

Los adaptadores AC/DC básicamente están formados por un circuito transformador, el cual convierte el voltaje AC de entrada en otro voltaje AC mucho menor, y un circuito rectificador, el cual convierte ese voltaje AC ya transformado en un voltaje DC, que será el voltaje final de salida. Todos los adaptadores incorporan una etiqueta impresa que informa tanto del rango de valores en el voltaje AC de entrada con el que son capaces de trabajar (además de la frecuencia de la señal AC admitida) como del valor del voltaje DC y de la intensidad máxima que ofrecen como salida. Por ejemplo, la imagen siguiente corresponde a un adaptador AC/DC que admite voltajes AC de entrada entre 100 V y 240 V a una frecuencia de 50 o 60Hz (por tanto, compatible con la red eléctrica española) y aporta un voltaje DC de salida de 9 V (y una intensidad máxima de 1 A).



Podemos clasificar los adaptadores según si son “regulados” (es decir, si incorporan un regulador de voltaje en su interior) o no. Un regulador de voltaje es un dispositivo (o un conjunto de ellos) que, estando sometido a un determinado voltaje de entrada relativamente fluctuante, es capaz de generar un voltaje de salida normalmente menor, mucho más estable, constante y controlado.

Por tanto, los adaptadores regulados proporcionan un voltaje de salida muy concreto y constante, que es igual al mostrado en su etiqueta. Lo que sí puede variar (hasta un máximo mostrado también en la etiqueta) es la intensidad de corriente ofrecida, ya que esta depende en cada momento de las necesidades del circuito alimentado.

Los adaptadores no regulados, en cambio, no poseen ningún mecanismo de estabilización y proporcionan un voltaje de salida cuyo valor puede llegar a ser diferente en varios voltios al mostrado en la etiqueta. Este tipo de adaptadores ciertamente reducen el voltaje de entrada a un valor de salida menor, pero el valor concreto de este voltaje de salida depende en buena parte del consumo eléctrico (medido en amperios o miliamperios) realizado en ese momento particular por el circuito alimentado.

Expliquemos esto: a medida que el circuito consume más intensidad de corriente, el voltaje de salida (inicialmente bastante más elevado que el valor nominal marcado en la etiqueta del adaptador) se va reduciendo cada vez más hasta llegar a su valor nominal solo cuando el circuito consume la máxima intensidad que el adaptador es capaz de ofrecer, (cuyo valor está indicado también en la etiqueta impresa, como ya sabemos). Si el circuito sigue aumentando su consumo y supera esa intensidad máxima, el voltaje ofrecido por el adaptador seguirá disminuyendo y llegará a ser menor que el nominal, circunstancia en la que se corre el riesgo de dañar el adaptador (y de rebote, el circuito alimentado). Este comportamiento es fácil comprobarlo con un multímetro, tal como veremos en un apartado posterior de este mismo capítulo.

La principal razón de la existencia de los adaptadores no regulados es su precio: son más baratos y además están disponibles en una gran variedad de formas y rangos de valores de uso. Generalmente, los adaptadores que se conectan directamente a los enchufes de la red (en forma de “verruga de pared”, o “wall-wart”) suelen ser no regulados. Los que, como los usados en los computadores portátiles, tienen forma de caja rectangular de la que salen el cable para enchufar a la red eléctrica y el cable para conectar el aparato, suelen ser regulados. De todas formas, independientemente del encapsulado del adaptador, una regla que por norma general se suele cumplir es que si el adaptador admite un rango de voltaje de entrada muy amplio (de 100 a 240 V, por ejemplo), entonces seguramente es regulado.

Un adaptador (regulado) que nos puede venir bien para nuestros proyectos de Arduino es por ejemplo el producto nº 63 de Adafruit: es un adaptador compatible con la red eléctrica española, y genera un voltaje de salida de 9 V y una corriente máxima de 1 A, lo que lo hace perfectamente compatible con las placas Arduino (además, su clavija de tipo “jack” es de 5,5 mm/2,1 mm, tal como es el zócalo de dichas placas). También nos podría servir el producto nº 798. Incluso podemos adquirir adaptadores que, en vez de la clavija jack de 5,5 mm/2,1 mm, ofrezcan una conexión de tipo USB: un ejemplo sería el producto con código FIT0197 de DFRobot, el cual proporciona 5 V y una corriente máxima de 1 A.

Si necesitáramos en cambio más voltaje de salida (porque en nuestro circuito tenemos dispositivos que realizan un mayor consumo, como por ejemplo muchos tipos de motores), tendríamos que utilizar un adaptador (regulado) como por ejemplo el producto nº 352 de Adafruit (el cual ofrece un voltaje de salida de 12 V y corriente máxima de 5 A) o el nº 658 (5 V y 10 A). Hay que tener la precaución, no obstante, de no utilizar un adaptador que ofrezca un voltaje o intensidad de salida mayor que el circuito sea capaz de admitir: si conectáramos por ejemplo una placa Arduino a estos dos últimos adaptadores, la placa se quemaría.

COMPONENTES ELÉCTRICOS

Resistencias



Un resistor o resistencia es un componente electrónico utilizado simplemente para añadir, como su nombre indica, una resistencia eléctrica entre dos puntos de un circuito. De esta manera, y gracias a la Ley de Ohm, podremos distribuir según nos convenga diferentes tensiones y corrientes a lo largo de nuestro circuito.

Debido al pequeño tamaño de la mayoría de resistores, normalmente no es posible serigrafiar su valor sobre su encapsulado, por lo que para conocerlo debemos saber interpretar una serie de líneas de colores dispuestas a lo largo de su cuerpo. Normalmente, el número de líneas de colores son cuatro, siendo la última de color dorado o bien plateado (aunque puede ser de otros colores también). Esta línea dorada o plateada indica la tolerancia de la resistencia, es decir: la precisión de fábrica que esta nos aporta. Si es de color dorado indica una tolerancia del ±5% y si es plateada una del ±10% (otros colores –rojo, marrón, etc. – indican otros valores). Por ejemplo, una resistencia de 220 Ω con una franja plateada de tolerancia, tendría un valor posible entre 198 Ω y 242 Ω (es decir, 220 Ω ±10%); obviamente, cuanto menor sea la tolerancia, mayor será el precio de la resistencia.

Las otras tres líneas de colores indican el valor nominal de la resistencia. Para interpretar estas líneas correctamente, debemos colocar a nuestra derecha la línea de tolerancia, y empezar a leer de izquierda a derecha, sabiendo que cada color equivale a un dígito diferente (del 0 al 9). La primera y segunda línea las tomaremos cada una como el dígito tal cual (uno seguido del otro) y la tercera línea representará la cantidad de ceros que se han de añadir a la derecha de los dos dígitos anteriores. La tabla para conocer el significado numérico de los posibles colores de una resistencia es la siguiente:

Color de la banda	Valor equivalente (en la 1ª y 2ª banda)	Multiplicador (valor de la 3ª banda)
Negro	0	$x10^0$ =1
Marrón	1	$x10^1$ =10
Rojo	2	$x10^2$ =100
Naranja	3	$x10^3$ =1000
Amarillo	4	$x10^4$ =10000
Verde	5	$x10^5$ =100000
Azul	6	$x10^6$ =1000000
Violeta	7	$x10^7$ =10000000
Gris	8	$x10^8$ =100000000
Blanco	9	$x10^9$ =1000000000

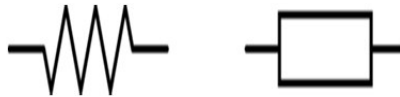
Por ejemplo, si tenemos una resistencia con las líneas de colores “rojo-verde-naranja”, podremos consultar la tabla para deducir que tendremos una resistencia de 25 · 1000 = 25 KΩ. Otro ejemplo: si tenemos una resistencia con las líneas de colores “marrón-negro-azul”, tendremos entonces una resistencia de 10 · 1000000 = 10 MΩ.

También nos podemos encontrar con resistencias que tengan cinco líneas impresas: en ese caso, su interpretación es exactamente igual, solo que en vez de dos disponemos de tres líneas para indicar los tres primeros dígitos del valor de la resistencia, siendo la cuarta la que representa el multiplicador y la quinta la tolerancia. Algunas resistencias incluso tienen hasta seis líneas impresas (son las más precisas, pero en nuestros proyectos pocas veces las necesitaremos); en ese caso, lo único que cambia es que aparece una sexta línea a la derecha de la línea de la tolerancia indicando un nuevo dato: el coeficiente de temperatura de la resistencia, el cual nos informa sobre cuánto varía el valor de esa resistencia dependiendo de la temperatura ambiente (medida en ppm/ºC, donde 10000 ppm = 1%). Otras resistencias (especialmente las de reducido tamaño, como las soldadas directamente a la superficie de una placa de circuito impreso) utilizan, en lugar de colores, una secuencia de tres dígitos para indicar las dos primeras cifras del valor de la resistencia y su multiplicador.

Ha de quedar claro que aunque para conocer el orden de las franjas y leer el valor de una resistencia hemos de colocar esta en un sentido determinado, los resistores no tienen polaridad. Esto quiere decir que a la hora de conectarlo en un circuito, es indiferente conectar sus dos terminales en un sentido o del revés.

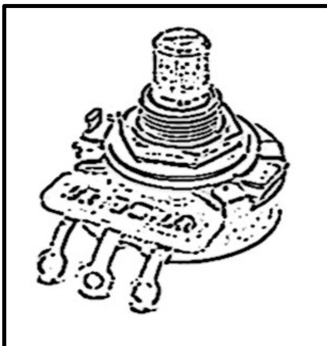
Por otro lado, además de conocer el valor resistivo que aportan estos componentes, también hemos de tener en cuenta la intensidad de corriente que pueden soportar como máximo sin fundirse. Para ello, el fabricante nos deberá proporcionar siempre un dato: la potencia máxima que la resistencia es capaz de disipar en forma de calor, valor que está directamente relacionado con su tamaño. Las resistencias más utilizadas en la electrónica son las de 1/4W, 1/2W y 1W (siendo la de 1/4W es la más pequeña y la de 1W es la más grande de las tres). En este sentido, para conocer qué tipo de resistencia nos interesará utilizar en nuestros circuitos, debemos utilizar la fórmula ya conocida de $P = V \cdot I$ (donde V es la diferencia de potencial presente entre los extremos de la resistencia e I es la corriente que circula por ella), o bien alguna de las fórmulas equivalentes, como $P = I^2 \cdot R$ o $P = V^2/R$ (donde R es el valor de la resistencia propiamente dicha). De cualquiera de estas maneras, obtendremos la potencia que debe ser capaz de disipar nuestra resistencia, con lo que ya tendremos el criterio para elegirla. No es mala idea utilizar una resistencia cuyo poder disipador sea aproximadamente el doble del resultado obtenido para no sufrir posibles sobrecalentamientos.

Los símbolos utilizados en el diseño de los circuitos eléctricos para representar una resistencia pueden ser dos:



donde el de la derecha es el estándar normalizado por la "International Electrotechnical Commission" (IEC), aunque el de la izquierda sigue siendo ampliamente utilizado actualmente.

Potenciómetros



Un potenciómetro es una resistencia de valor variable. Podemos darnos cuenta de su gran utilidad con un ejemplo muy simple: si suponemos que tenemos una fuente de alimentación que genera un determinado voltaje estable, y tenemos presente la Ley de Ohm ($V = I \cdot R$), podemos ver que si aumentamos de valor la resistencia R, a igual voltaje la intensidad de corriente que pasará por el circuito inevitablemente disminuirá. Y al contrario: si disminuimos el valor de R, la corriente I aumentará. Si esta variación de R la podemos controlar nosotros a voluntad, podremos alterar como queramos la corriente que circula por un

circuito. De hecho, un uso muy habitual de los potenciómetros es el de hacer de divisores de tensión progresivos, con lo que podremos, por poner un ejemplo, encender o apagar paulatinamente una luz a medida que vayamos cambiando el valor de R.

Un potenciómetro dispone físicamente de tres patillas: entre las dos de sus extremos existe siempre un valor fijo de resistencia (el máximo, de hecho), y entre cualquiera de esos extremos y la patilla central tenemos una parte de ese valor máximo. Es decir: la resistencia máxima que ofrece el potenciómetro entre sus dos extremos no es más que la suma de las resistencias entre un extremo y la patilla central (llamémosla R1), y entre la patilla central y el otro extremo (llamémosla R2). De aquí se puede pensar que un potenciómetro es equivalente a dos resistencias en serie, pero la gracia está en que en cualquier momento podremos modificar el estado de la patilla central para conseguir aumentar la resistencia de R1 (disminuyendo como consecuencia la resistencia R2, ya que el valor total máximo sí que permanece constante) o bien al contrario, para conseguir disminuir la resistencia de R1 (aumentando por lo tanto la resistencia R2 automáticamente).

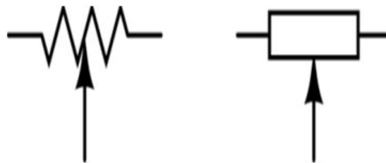
La manera concreta de alterar el estado de la patilla central del potenciómetro puede variar y suele depender de su encapsulamiento físico, pero por lo general suele consistir en el desplazamiento de un cursor manipulable conectado a dicha patilla. Podemos encontrarnos potenciómetros de movimiento rotatorio como los del control de volumen de la mayoría de altavoces, o de movimiento rectilíneo como los que se utilizan en las mesas de mezcla de sonido, entre otros. En la imagen mostrada anteriormente se puede ver uno de movimiento rotatorio.

También existen potenciómetros de tipo digital: estos son chips que constan de diferentes patillas a través de las que se puede controlar mediante pulsos eléctricos los valores extremos de la resistencia y su valor intermedio. Un ejemplo es el componente DS1669 de Maxim.

La clasificación más interesante, no obstante, viene a la hora de distinguir el comportamiento que tiene un potenciómetro en el momento que modificamos el estado de su patilla central (es decir, cuando “es movida”). Si el potenciómetro tiene un comportamiento llamado “lineal”, la alteración del valor de su resistencia es siempre directamente proporcional al recorrido de la patilla central: es decir, si desplazamos por ejemplo la patilla un 30% se aumentará/disminuirá la resistencia un 30% también. Por el contrario, si el potenciómetro tiene un comportamiento “logarítmico”, la alteración del valor de su resistencia será muy leve al principio del recorrido de la patilla (y por tanto, habrá que realizar un gran desplazamiento de esta

para obtener un cambio apreciable de resistencia) pero a medida que se siga realizando más recorrido de la patilla, la alteración de la resistencia cada vez será proporcionalmente mayor y mayor, hasta llegar a un punto donde un leve desplazamiento producirá una gran cambio en la resistencia. Los potenciómetros logarítmicos son empleados normalmente para el audio, ya que el ser humano no oye de manera lineal: para experimentar por ejemplo una sensación acústica de "el doble de fuerte", es necesario que el volumen físico del sonido sea unas diez veces mayor.

Los símbolos que se pueden utilizar en el diseño de circuitos electrónicos para representar un potenciómetro son los siguientes:



Otras resistencias de valor variable

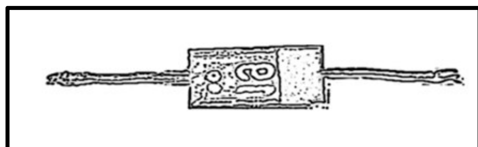
Los potenciómetros son resistencias que cambian su valor según nuestra voluntad. Pero también existen resistencias que cambian su valor según condicionantes ambientales externos.

Por ejemplo, los fotorresistores (también llamados LDRs –del inglés "Light Dependent Resistor" – o también "celdas CdS" –por el material con el que habitualmente están fabricadas: sulfuro de cadmio–) son resistencias que varían según la cantidad de luz que incide sobre ellos, por lo que se pueden utilizar como sensores de luz. Otro ejemplo son los termistores: resistencias que cambian su valor según varíe la temperatura ambiente, por lo que se pueden utilizar como sensores de temperatura. Otro ejemplo son los sensores de fuerza/presión (también llamados FSRs –del inglés "Force-Sensing Resistor" –), que son resistencias cuyo valor depende de la fuerza/presión a la que son sometidas, o los sensores de flexión: resistencias cuyo valor varía según lo que sean dobladas físicamente, etc.

En el capítulo dedicado a los sensores hablaremos más extensamente de las diferentes características de estos componentes y de sus posibles usos prácticos. Si lo que se desea es conocer el símbolo esquemático correspondiente a un tipo concreto de resistencia variable (LDR, termistor, etc.), recomiendo consulta la siguiente página

web, la cual incluye la mayoría de símbolos electrónicos existentes, clasificados y ordenados por categorías: http://www.simbologia-electronica.com

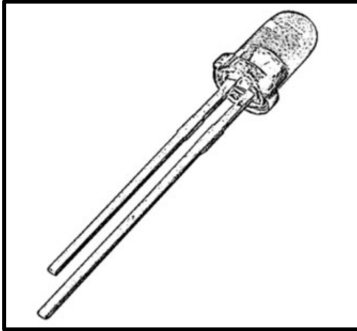
Diodos y LEDs



El diodo es un componente electrónico con dos extremos de conexión (o "terminales") que permite el paso libre de la corriente eléctrica solamente en un sentido, bloqueándolo si la corriente fluye en el sentido contrario. Este hecho hace que el diodo tenga dos posiciones posibles: a favor de la corriente (llamada "polarización directa") o en contra ("polarización inversa"). Por tanto, a la hora de utilizarlo en nuestros circuitos, debemos de tener en cuenta que la conexión de sus dos terminales se realice en el sentido deseado. Normalmente, los fabricantes nos indicarán cuál es el terminal que ha de conectarse al polo negativo (suponiendo polarización directa) mediante una marca visible cerca de este pintada en el cuerpo del diodo. En la imagen mostrada, esta marca es la gruesa franja blanca a la derecha del cuerpo del diodo, por lo que el "terminal negativo" será, en este caso, el de la derecha. Técnicamente (siempre suponiendo polarización directa) a ese "terminal negativo" se le llama "cátodo", y al "terminal positivo" se le llama "ánodo".

El diodo se puede utilizar para muchos fines: un uso común es el de rectificador (para convertir una corriente alterna en continua), pero en nuestros circuitos lo usaremos sobre todo como un elemento suplementario conectado a algún otro componente para evitar que este se dañe si la alimentación eléctrica se conecta por error con la polaridad al revés.

Es costumbre conectar un divisor de tensión (es decir, una resistencia en serie) a uno de los terminales del diodo (es indiferente si es el ánodo o el cátodo) para evitar que sea este precisamente el que se funda al recibir más tensión de la que pueda soportar. Para calcular el valor de esta resistencia, debemos tener en cuenta la intensidad que debe pasar por el diodo (I), la tensión que existiría entre sus terminales si no pusiéramos ninguna resistencia (V) y la tensión entre sus terminales que queremos conseguir para evitar daños (V_{DIO}); una vez conocido estos valores podemos calcular la resistencia adecuada usando la Ley de Ohm, así: $R = (V-V_{DIO})/I$. Por otro lado, la potencia disipada por esa resistencia podríamos calcularla mediante la fórmula $P = (V-V_{DIO}) \cdot I$.



Un “Light Emitting Diode” (LED) es, como su nombre indica, un diodo que tiene una característica peculiar: emite luz cuando la corriente eléctrica lo atraviesa. De hecho, lo hace de forma proporcional: a más intensidad de corriente que lo atraviesa, más luz emite.

Ya que no deja de ser un tipo concreto de diodo, también puede ser conectado en polarización directa o inversa, teniendo en cuenta que solo se iluminarán si están conectados en polarización directa. Por ello, cuando diseñemos nuestros circuitos hay que seguir teniendo la precaución de conectar cada terminal del LED en la polaridad adecuada. No obstante, como a un LED no se le puede pintar una marca encima, la manera de distinguir el ánodo (“terminal positivo” en polarización directa) del cátodo (el “terminal negativo” en polarización directa) es observando su longitud: el ánodo es de una longitud más larga que el cátodo.

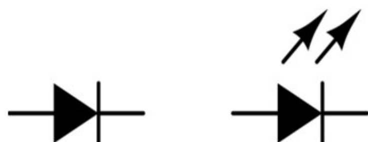
Es igualmente muy recomendable conectar una resistencia en serie a un LED para limitar la intensidad de corriente que lo atraviesa y así mantenerla por debajo del valor máximo más allá del cual el LED puede dañarse. Para calcular qué resistencia debemos colocar, podemos utilizar la fórmula mencionada en un par de párrafos anteriores. Hemos de saber que normalmente la intensidad que suele venir bien para el funcionamiento óptimo de un LED es de unos 15 mA, y que la tensión V_{DIO} apropiada varía según el color del LED: va de 3 V a 3,6 V para el ultravioleta (UV), blanco o azul, de 2,5 V a 3 V para el verde, de 1,9 V a 2,4 V para el rojo, naranja, amarillo o ámbar y de 1 V a 1,5 para el infrarrojo. A partir de aquí, el cálculo es sencillo. No obstante, para mayor seguridad es recomendable consultar siempre las especificaciones que proporciona el fabricante (el “datasheet” del componente) para conocer toda la información necesaria sobre intensidades y tensiones máximas soportadas.

Además de por sus diferentes colores (consecuencia del material de fabricación usado, diferente para cada tipo de LED), podemos clasificar estos componentes según si emiten la luz de forma difusa o clara. Los primeros (que normalmente tienen un tamaño de 3 mm de diámetro) se suelen utilizar para indicar presencia, ya que emiten una luz suave y uniforme que no deslumbra y que puede verse bien desde cualquier ángulo. Los segundos (que normalmente tienen un tamaño de 5 mm de diámetro) sirven para irradiar en una dirección muy concreta con luz directa y potente, por lo que no se ven bien en todos los ángulos pero iluminan mucho más que los otros. En cualquier caso, sea un LED de tipo difuso o claro, para

saber en términos cuantitativos lo “brillante” que es su luz, será necesario conocer la cantidad de milicandelas (mcd) que ese LED concreto es capaz de emitir; este dato lo debe ofrecer el fabricante en el datasheet.

Finalmente, debemos tener en cuenta al menos tres aspectos más a la hora de usar diodos y LEDs en nuestros proyectos: la corriente máxima que puede atravesar el diodo en polarización directa sin que este se funda debido al calor generado por la potencia disipada; el voltaje de ruptura (cuando los diodos están conectados en polarización inversa hemos dicho que no dejan pasar el flujo de la corriente pero dicho comportamiento es así solamente mientras al diodo se le aplique un voltaje menor del llamado “voltaje de ruptura”) y la caída de tensión en polarización directa (como cualquier otro dispositivo electrónico, los diodos poseen una resistencia interna que provoca la existencia de una determinada diferencia de potencial entre sus terminales).

El símbolo que se suele utilizar en el diseño de circuitos electrónicos para representar un diodo estándar es el mostrado a la izquierda en la imagen siguiente, y el de un LED es el de la derecha.



El vértice del triángulo secante con la línea perpendicular de ambos símbolos representa el cátodo. Existen otros símbolos similares que representan diodos más específicos (como los de tipo Zener o Schottky, entre otros) pero no los veremos.

Condensadores

El condensador es un componente cuya función básica es almacenar carga eléctrica en cantidades limitadas, de manera que esta se pueda utilizar en ocasiones muy puntuales a modo de “fuente de alimentación alternativa”.

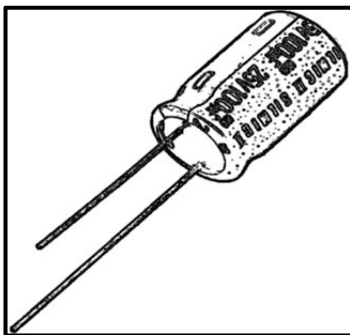
La capacidad (C) de un condensador es su característica más importante y se puede definir como la relación –normalmente de un valor constante– que existe entre la cantidad de carga eléctrica (Q) que almacena en un momento determinado y el voltaje (V) que se le está aplicando en ese mismo momento. Concretamente, se define así: **C = Q/V**.

De la fórmula anterior podemos deducir varias cosas: la primera es que un condensador con mayor capacidad que otro almacenará más carga bajo el mismo potencial. La segunda es que un condensador con una determinada capacidad almacenará más carga cuanto mayor sea el voltaje aplicado (aunque en este sentido hay que tener en cuenta que todo condensador tiene un voltaje de trabajo máximo –que suele venir impreso en el cuerpo del propio condensador– más allá del cual se puede dañar, por lo que siempre se tendrá un máximo de carga almacenable).

La capacidad se mide en faradios (F), aunque la mayoría de condensadores con los que trabajaremos tienen una capacidad mucho menor (del orden de los microfaradios o incluso nanofaradios). Dependiendo del tamaño del condensador, puede ser que el valor de su capacidad no pueda ser serigrafiado tal cual sobre su cuerpo; en esos casos, se suele utilizar una secuencia de tres dígitos para indicar las dos primeras cifras del valor de la capacidad y luego su multiplicador. Por ejemplo, un condensador con el número "403" impreso, querrá decir que tiene una capacidad de $40 \cdot 10^3 = 40 \cdot 1000 = 40000$ F.

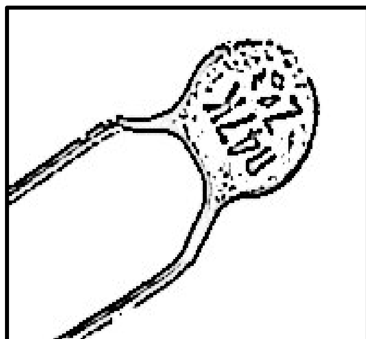
Al igual que ocurría con las resistencias, los condensadores pueden ser conectados en serie o en paralelo para conseguir un circuito con una capacidad equivalente. En concreto, si dos condensadores (C1 y C2) se conectan en serie, la capacidad equivalente es **C = (C1·C2)/(C1+C2)**; es decir, menor que cualquiera de la de los condensadores individuales. Si los dos condensadores se conectan en paralelo, la capacidad equivalente es **C = C1+C2** ; es decir, obtenemos una capacidad total mayor.

Un condensador completamente cargado, en circuitos de corriente DC, actúa como un interruptor abierto.



Podemos clasificar los tipos de condensadores según si tienen o no polarización. Los condensadores polarizados son los que se han de conectar al circuito respetando el sentido de la corriente. Es decir, tienen un terminal "negativo" que siempre deberá conectarse al polo negativo del circuito, y otro terminal "positivo" que siempre deberá conectarse al polo positivo. Dicho de otra forma: su conexión se ha de realizar siempre en polarización directa. Es por ello que este tipo de condensadores no son válidos para ser usados en corriente alterna; de hecho, al conectarlos en polarización inversa son destruidos. Para distinguir un terminal del otro, podemos fijarnos en una franja pintada sobre el cuerpo del condensador, la cual indicará siempre el terminal

negativo (de forma similar a lo que ocurría con los diodos). Otra pista que podemos usar para lo mismo es fijarnos en que el terminal negativo es más corto que el positivo (al igual que ocurría con los LEDs). Como características más reseñables, podemos decir que los condensadores polarizados tienen por norma una capacidad bastante elevada (de un valor mayor de 1 microfaradio), y, teniendo en cuenta el modo de fabricación, suelen ser de tipo electrolítico (como el mostrado en la imagen) o de tantalio.



Los condensadores unipolares (no polarizados) pueden conectarse al circuito en ambos sentidos indiferentemente (al igual que las resistencias, por ejemplo). Pueden estar fabricados de muchos materiales, pero los más comunes son los de tipo cerámico (como el mostrado en la imagen). Por lo general, suelen tener una capacidad bastante menor que los condensadores polarizados.

Los condensadores muchas veces se utilizan en los circuitos para proporcionar la llamada “alimentación de desvío” o “desacople” (en inglés, “by-pass” o “decoupling”). Esta alimentación es necesaria cuando un componente que normalmente no requiere de mucha intensidad de corriente para funcionar, ha de realizar de forma puntual un consumo tan elevado de electricidad que la fuente de alimentación ordinaria no es capaz de ofrecerla con la celeridad suficiente. Esto normalmente ocurre cuando dicho componente (un microcontrolador, por ejemplo) pasa de estar en estado desactivado a activado; en ese momento es cuando el condensador proporciona rápidamente la corriente transitoria necesaria al “soltar” la carga eléctrica que mantenía almacenada. Con este sistema de alimentación “alternativa”, se logra dar una rápida respuesta al pico de consumo del componente mientras que la fuente de alimentación puede ir volviendo a cargar otra vez el condensador (a un ritmo menor) de cara al próximo pico. Para utilizar un condensador que realice esta función, uno de sus terminales ha de conectarse lo más cerca posible de la entrada de alimentación del componente a “gestionar” (cuanto más lejos, menor es el efecto) y el otro terminal a la tierra del circuito. Valores típicos de un condensador “by-pass” son 0,1 µF o 0,01µF.

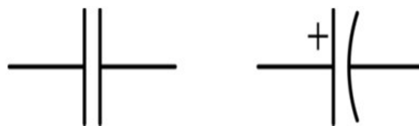
Otro uso muy frecuente de los condensadores es la eliminación del “ruido” de la señal de alimentación DC. Es decir, aunque una fuente sea etiquetada nominalmente como de 9 V, por ejemplo, en la realidad nunca ofrecerá esos 9 V exactos, sino que ese valor irá sufriendo variaciones más o menos amplias y

aleatorias alrededor de su valor nominal. Por tanto, además de una alimentación DC tenemos siempre una pequeña alimentación AC alrededor de aquella. Dependiendo de la magnitud de esta alimentación AC (es decir, de las variaciones alrededor de los 9 V del ejemplo), podríamos llegar a tener un problema en nuestro circuito, porque aparecerían efectos de corrientes de tipo AC que no deseamos. Un condensador es capaz de estabilizar esas variaciones, permitiendo obtener de la fuente un valor de tensión más constante, gracias a que es capaz de regular convenientemente la carga que "suelta" o "acumula" en función del voltaje fluctuante al que es sometido.

Para utilizar un condensador que realice esta función de estabilización de la señal (los cuales se suelen llamar "condensadores de filtro"), uno de sus terminales se ha de conectar al borne positivo de la fuente de alimentación y el otro terminal se ha de conectar al borne negativo. Un valor típico de un condensador de filtro es 0,1 µF. A mayores frecuencias de corriente AC, se necesitan condensadores de menor capacidad. Incluso se pueden conectar varios condensadores en paralelo de diferente capacidad, para filtrar diferentes frecuencias (pero el cálculo concreto se sale de los objetivos de este libro).

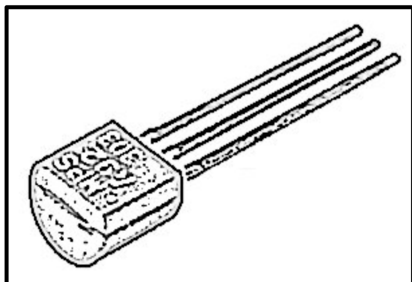
Realmente, los condensadores son usados en multitud de aplicaciones: como baterías y memorias por su cualidad de almacenar carga, para realizar descargas rápidas (como la luz "flash" de una cámara fotográfica), para mantener corrientes estables (como por ejemplo las generadas por un rectificador), para evitar caídas de corriente puntuales en los circuitos (es decir, la función de "by-pass"), para aislar partes de un circuito (cuando están completamente cargados), etc.

Los símbolos utilizados en el diseño de los circuitos eléctricos para representar un condensador pueden ser dos:



La representación de la izquierda es la de un condensador unipolar, y la de la derecha es la de un condensador polarizado. En este último, la línea recta simboliza el polo positivo (a veces el signo "+" se omite y/o la línea recta se pinta más gruesa) y la línea curva simboliza el polo negativo.

Transistores



Un transistor es un dispositivo electrónico que restringe o permite el flujo de corriente eléctrica entre dos contactos según la presencia o ausencia de corriente en un tercero. Puede entenderse como una resistencia variable entre dos puntos, cuyo valor es controlado mediante la aplicación de una determinada corriente sobre un tercer punto.

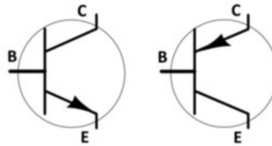
Los transistores se suelen utilizar como amplificadores de corriente, ya que con una pequeña corriente recibida a través de su terminal de control permiten la circulación de una intensidad muy grande (proporcional a aquella, hasta un máximo) entre sus dos terminales de salida. Otro uso muy frecuente es el de ser conmutadores de corriente, ya que si su terminal de control no recibe ninguna intensidad de corriente, por entre los dos terminales de salida no fluye ninguna corriente tampoco y se abre el circuito.

Existen dos grandes categorías de transistores según su tecnología de fabricación y funcionamiento: los transistores de tipo bipolar (llamados comúnmente BJT, del inglés “Bipolar Junction Transistor”) y los transistores de tipo efecto de campo (llamados comúnmente FET, del inglés “Field Effect Transistor”).

Los transistores BJT disponen de tres patillas físicas y cada una tiene un nombre específico: “Colector”, “Base” y “Emisor”. La “Base” hace de “terminal de control” y el “Colector” y el “Emisor” son los “terminales de salida”. De todas maneras, dependiendo de cómo se utilicen y conecten estas tres patillas, podemos clasificar a su vez los transistores BJT en dos tipos, los NPN –los más habituales– y los PNP. En el caso de los NPN, si aplicamos cierta corriente (por lo general muy baja) de la Base al Emisor, el Emisor actuará como una válvula que regulará el paso de corriente desde el Colector hacia el propio Emisor. En el caso de los PNP, si aplicamos cierta corriente (por lo general muy baja) del Emisor a la Base, el Emisor actuará como una “válvula” que regulará el paso de corriente desde el propio Emisor hacia el Colector.

Físicamente, los transistores BJT pueden ser muy diferentes, pero el mostrado en la figura de la página anterior es un encapsulado típico, en el cual las tres patillas se corresponden con el Colector, Base y Emisor (aunque para saber cuál es cuál se

deberá consultar el datasheet del dispositivo). Los símbolos utilizados en el diseño de los circuitos eléctricos para representar los transistores NPN y PNP son:



Cuando el transistor se utiliza como amplificador, la intensidad de corriente que circula del Emisor al Colector (si es PNP) o del Colector al Emisor (si es NPN) puede llegar a ser decenas de veces mayor que la corriente que apliquemos al terminal de entrada Base-Emisor. Este factor de proporcionalidad entre la intensidad que pasa por la Base y la que pasa por el Colector (es decir, el factor de amplificación o ganancia) depende del tipo de transistor y ha de venir descrito en sus especificaciones técnicas (el "datasheet") con el nombre de β o $h_{FE.}$

Más en concreto, podemos distinguir tres "modos de funcionamiento" en un transistor típico:

El modo de corte: se produce cuando la corriente que fluye por la Base es próxima a 0. En ese caso, no circula corriente por el interior del transistor, impidiendo así el paso de corriente tanto por el Emisor como por el Colector (es decir, el transistor se comporta como un interruptor abierto).

El modo de saturación: se produce cuando la corriente que fluye por el Colector es prácticamente idéntica a la que fluye por el Emisor (momento en el cual, de hecho, estas se aproximan al valor máximo de corriente que puede soportar el transistor en sí). Es decir, el transistor se comporta como una simple unión de cables, ya que la diferencia de potencial entre Colector y Emisor es muy próxima a cero. Este modo se da cuando la intensidad que circula por la Base supera un cierto umbral.

El modo activo: se produce cuando el transistor no está ni en su modo de corte ni en su modo de saturación (es decir, en un modo intermedio). Es en este modo cuando la corriente de circula por el Colector depende principalmente de la corriente de la Base y de β (la ganancia de corriente). Concretamente, se cumple que $I_c = \beta \cdot I_b$ e $I_e = I_c + I_b$, donde I_c es la corriente que fluye por el Colector, I_b la que fluye por la Base y I_e la que fluye por el Emisor.

Como se puede ver, el modo activo es el interesante para usar el transistor como un amplificador de señal, y los modos de corte y saturación son los interesantes para usar el transistor como un conmutador que represente el estado lógico BAJO y ALTO, respectivamente.

Por su lado, los transistores FET cumplen la misma función que los BJT (amplificador o conmutador de la corriente, entre otras), pero sus tres terminales se denominan (en vez de Base, Emisor y Colector): Puerta (identificado como "G", del inglés "Gate"), Surtidor (S) y Drenador (D). El terminal G sería el "equivalente" a la Base en los BJT, pero la diferencia está en que el terminal G no absorbe corriente en absoluto (frente a los BJT donde la corriente atraviesa la Base pese a ser pequeña en comparación con la que circula por los otros terminales). El terminal G más bien actúa como un interruptor controlado por tensión, ya que (y aquí está la clave del funcionamiento de este tipo de transistores) será el voltaje existente entre G y S lo que permita que fluya o no corriente entre S y D.

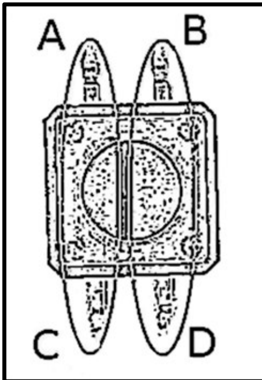
Así como los transistores BJT se dividen en NPN y PNP, los de efecto de campo (FET) son también de dos tipos: los de "canal n" y los de "canal p", dependiendo de si la aplicación de una tensión positiva en la puerta pone al transistor en estado de conducción o no conducción, respectivamente.

Por otro lado, los transistores FET también se pueden clasificar a su vez dependiendo de su estructura y composición interna. Así tenemos los transistores JFET (Junction FET), los MOS-FET (Metal-Oxide-Semiconductor FET) o MIS-FET (Metal-Insulator-Semiconductor FET), entre otros. Cada uno de estos tipos tiene diferentes características específicas que los harán más o menos interesantes dependiendo de las necesidades del circuito, y cada uno tiene un símbolo esquemático diferente.

En general, los transistores FET se suelen utilizar más que los BJT en circuitos que consumen gran cantidad de potencia.

Pulsadores

Ya sabemos que un interruptor es un dispositivo con dos posiciones físicas: en la posición de "cerrado" se produce la conexión de dos terminales (lo que permite fluir a la corriente a través de él) y en la posición de "abierto" se produce la desconexión de estos dos terminales (y por tanto se corta el flujo de corriente a través de él). En definitiva, que un interruptor no es más que un mecanismo constituido por un par de contactos eléctricos que se unen o separan por medios mecánicos.



Un pulsador (en inglés, “pushbutton”) no es más que un tipo de interruptor en el cual se establece la posición de encendido mediante la pulsación de un botón gracias a la presión que se ejerce sobre una lámina conductora interna. En el momento de cesar la pulsación sobre dicho botón, un muelle hace recobrar a la lámina su posición primitiva, volviendo a la posición de “abierto”.

Existe una gran variedad de pulsadores de muchas formas y tamaños diferentes, pero en los circuitos que realizaremos a lo largo de este libro utilizaremos unos pulsadores muy prácticos y relativamente pequeños (tienen un tamaño de tan solo un 1/4 de pulgada por cada lado) que se pueden adquirir en cualquier distribuidor listado en el apéndice A (por ejemplo, en Sparkfun es el producto nº 97) y que son los que habitualmente vienen en los kits de aprendizaje.

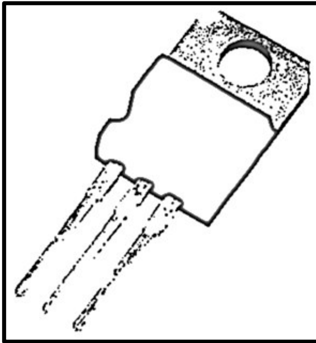
Estos pulsadores, no obstante, tienen una característica que conviene aclarar: tal como se ve en la imagen lateral, tienen cuatro patillas. Ello nos puede hacer pensar que tienen cuatro terminales, pero nada más lejos de la realidad: las dos patillas enfrentadas de cada lado están unidas internamente entre sí, por lo que funcionan como una sola. Es decir, las patillas A y C resaltadas en la imagen representan un solo punto eléctrico, y las patillas B y D representan otro punto eléctrico único. Por tanto, realmente, este pulsador solamente tiene dos terminales: A/C y B/D.

En la imagen siguiente se puede ver el símbolo eléctrico del pulsador en (a la izquierda estado abierto y a la derecha en estado cerrado), indicando además como ejemplo a qué patillas físicas corresponde cada terminal.



Hay que tener en cuenta que este tipo de pulsadores por lo general tienen una cantidad máxima bastante limitada de tensión y corriente que pueden resistir antes de quemarse: si se va a utilizar una alimentación más exigente, se deberán utilizar pulsadores más robustos. Para conocer los valores concretos de tensión, corriente (y presión sobre el botón, entre otras cosas) máximos que puede soportar un pulsador concreto, nos hemos de remitir a la documentación oficial que proporciona el fabricante para ese componente (el “datasheet”).

Reguladores de tensión



Un regulador de tensión es un componente electrónico que protege partes de un circuito (o un circuito entero) de elevados voltajes o de variaciones pronunciadas de este. Su función es proporcionar, a partir de un voltaje recibido fluctuante dentro de un determinado rango (el llamado “voltaje de entrada”), otro voltaje (el llamado “voltaje de salida”) regulado a un valor estable y menor. Esto lo puede conseguir aplicando la Ley de Ohm: gracias a su capacidad de elevar o disminuir (según el caso) la corriente interna que lo está atravesando en cada momento, pueden elevar o disminuir proporcionalmente su voltaje de salida. Así explicado podría parecer un simple divisor de tensión (y de hecho, su función es la misma), pero su mecanismo de regulación del voltaje de salida es mucho más sofisticado y fiable.

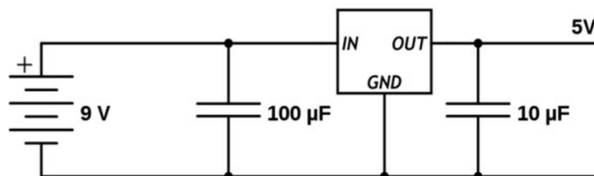
Son un elemento clave para conseguir una alimentación correcta y segura de los distintos componentes electrónicos de nuestros circuitos. Gracias a ellos, componentes que resultarían dañados si son sometidos a un voltaje demasiado elevado, pueden ser combinados en un mismo circuito por otros componentes más capaces y que requieran de una tensión mayor.

Existen muchos tipos y modelos de regulador, pero en nuestros circuitos normalmente utilizaremos pequeños componentes encapsulados llamados genéricamente reguladores LDO (del inglés “low-dropout”). El voltaje “dropout” es la diferencia entre el voltaje de entrada y el de salida. Este voltaje multiplicado por la corriente que lo atraviesa es gastado en forma de calor, por lo que cuanto menor sea el dropout, más eficiente será el regulador en términos de pérdida de energía. Por ejemplo, si tenemos un regulador LM8705 alimentado a 12 V que ofrece una tensión regulada de 5 V y se estima que la máxima corriente que consume nuestro circuito es de 1,2 A, podemos calcular la potencia perdida en forma de calor por este regulador con la fórmula ya conocida de P = V · I: (12 V-5V) · 1,2ª = 8,4 W. El valor obtenido es bastante elevado, pero afortunadamente, es el peor de los casos, ya que hemos considerado la corriente máxima consumida por el circuito; en general los microcontroladores y muchos dispositivos electrónicos consumen pulsos de corriente, por lo que la corriente promedio que suministra el regulador suele ser bastante menor, y por tanto, no se pierde tanta potencia (tanto calor) en el mismo.

Los reguladores LDO suelen tener tres patillas: una para recibir el voltaje de entrada, otra para ofrecer el voltaje de salida (que haría de "terminal positivo" para los componentes sensibles) y una tercera patilla conectada a la tierra común con la fuente de alimentación. Pero el orden y ubicación de cada patilla depende del modelo de regulador particular, así que se recomienda consultar la documentación técnica del fabricante para conocerla.

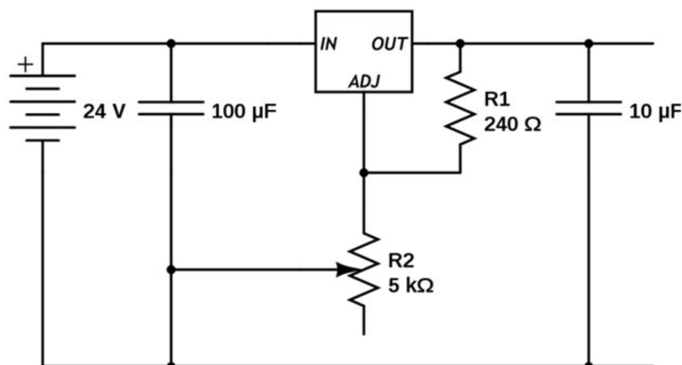
La familia de reguladores LDO más ampliamente utilizada en proyectos de electrónica doméstica es la LM78XX, donde "XX" indica el voltaje de salida. Así pues, muchas veces el modelo concreto que nos interesará es el LM7805, el cual puede recibir hasta entre 7 V y 35 V de entrada y puede generar una intensidad de salida máxima de 1 A. Otro modelo similar es el LM2940. Si queremos obtener una intensidad máxima de 0,1 A, debemos utilizar entonces el modelo LM78L05. Si queremos un voltaje de salida de 3,3, podemos usar el LM7803 o el LD1117V33, entre otros.

Sea cual sea el modelo de regulador, la mayoría de ocasiones veremos conectado a su patilla de entrada –y a tierra– un condensador "by-pass", y veremos conectado a su patilla de salida –y a tierra– un condensador de filtro. La razón es eliminar las posibles oscilaciones de la señal de entrada (provocadas por ejemplo por el repentino encendido de un elemento de alto consumo de nuestro circuito, como un motor) y las de la salida (optimizando así la tensión obtenida del regulador). Por tanto, el esquema de conexiones más común de un regulador típico (como por ejemplo el LM7805) es similar a este:



En la figura anterior se puede observar que hemos utilizado un condensador "by-pass" de 100 microfaradios y un condensador de filtro de 10 microfaradios. Estos valores suelen venir bien en la mayoría de circunstancias y son bastante "estándares". Se podrían conectar otros condensadores en paralelo a ambos lados de diferentes capacidades, para responder a mayores variaciones del voltaje de entrada y salida respectivamente, pero por lo general, no nos encontraremos en situaciones donde esto sea necesario.

Otro modelo común de regulador es el LM317, cuya característica más interesante es que es posible ajustar el voltaje de salida al que nosotros deseemos (concretamente entre 1,25 V y 37 V, con una intensidad mínima de salida de 1,5 V), y que es capaz de soportar voltajes de entrada de entre 3 V y 40 V. Para conseguir variar el voltaje de salida, disponemos de un circuito auxiliar conectado al regulador formado por una resistencia fija (R1) y un potenciómetro (R2), de la siguiente manera:

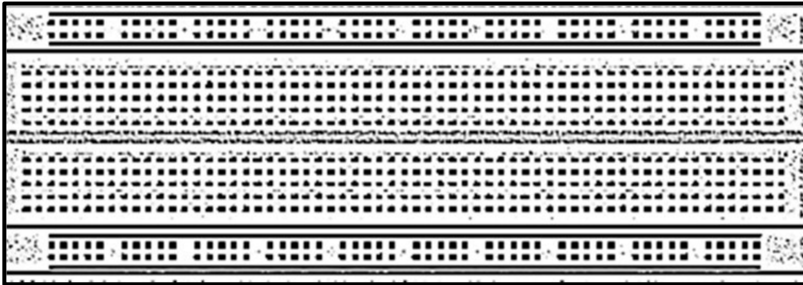


Usando el LM317 y el circuito mostrado en el esquema anterior, obtendremos un voltaje de salida dado por la expresión $V_{salida} = 1{,}25 \cdot (1 + R2/R1)$. Opcionalmente, al diseño anterior se le puede añadir un diodo para proteger el regulador contra posibles cortocircuitos en su entrada; para ello, deberíamos conectar el ánodo del diodo a la patilla de salida y el cátodo a la patilla de entrada.

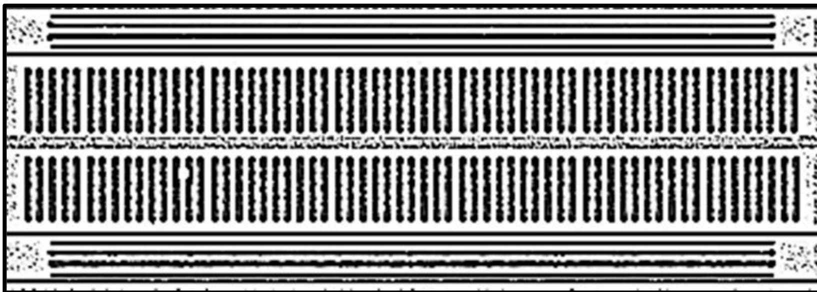
Placas de prototipado

Existen varios tipos de placas de prototipado. En este apartado estudiaremos solamente las llamadas “breadboards” (también conocidas como “protoboards”), las “perfboards” y las “stripboards”.

Una breadboard es una placa perforada con conexiones internas en la que podemos insertar las patas de nuestros componentes electrónicos tantas veces como queramos, realizando así las conexiones de nuestros circuitos sin tener la necesidad de soldar nada. El objetivo es poder montar prototipos rápidos pero completamente funcionales de nuestros diseños y poderlos modificar fácilmente cuando lo necesitemos. La siguiente imagen muestra la apariencia externa de una breadboard típica, que no deja de ser un conjunto de filas con agujeros.



Pero para poder conectar correctamente nuestros componentes a la breadboard, hemos de conocer primero cómo se estructuran sus propias conexiones internas. En este sentido, si observáramos su interior oculto bajo la superficie perforada, podríamos comprobar que está compuesto de muchas tiras de metal (normalmente cobre) dispuestas de la siguiente manera:



En la figura anterior se pueden distinguir básicamente tres zonas:

Buses: los buses se localizan en uno o ambos lados del protoboard. Allí se conectarán (en cualquiera de sus puntos) las fuentes de alimentación externas. Normalmente aparece pintada una línea roja que se suele utilizar para indicar el bus sometido al voltaje de entrada (es decir, donde insertaremos el borne positivo de la fuente) y una línea azul que representa el bus conectado a tierra (es decir, donde normalmente insertaremos el borne negativo). Todos los puntos del bus marcado con la línea roja son equivalentes porque están conectados entre sí y todos los puntos del bus marcado con la línea azul también lo son entre sí, pero ambos buses están aislados eléctricamente uno del otro.

Nodos: en la parte central del protoboard aparecen gran cantidad de agujeros. Su cantidad puede ser mayor o menor dependiendo del modelo (de hecho, el tamaño de la breadboard se indica por el número de filas y de columnas de agujeros que contiene: un tamaño típico es 10x64). Estos agujeros se usan para colocar los componentes y realizar las conexiones entre ellos. Tal como se puede observar en la figura anterior, las conexiones internas entre los agujeros están dispuestas en vertical. Lo más importante es comprender que cualquier agujero es completamente equivalente a otro que pertenezca a la misma conexión interna. Esto significa que al insertar una patilla de algún componente en un agujero, disponemos del resto de agujeros de su misma conexión interna para poder insertar en ellos una patilla de cualquier otro componente que queramos poner en contacto entre sí, tal como si los uniéramos directamente por un cable. A todos esos agujeros equivalentes conectados entre sí se les da el nombre en conjunto de "nodo". La manera más habitual de conectar dos nodos diferentes es enchufando los extremos de un cable en un agujero de cada nodo a unir.

Canal central: es la región localizada en el medio del protoboard, que separa la zona superior de la inferior. Se suele utilizar para colocar los circuitos integrados (esos componentes con forma de "cucarachas negras con patitas" también llamados "chips" o IC –del inglés "integrated circuits" –) de manera que pongamos la mitad de patitas en un lado del canal y la otra mitad en el otro lado. De esta manera, además de disponer así de varios agujeros de conexión por cada patita, una mitad del chip estará aislada eléctricamente de la otra (tal como debe ser).

También es frecuente el uso de "minibreadboards", especialmente pensadas para proyectos más compactos, ya que carecen de los buses de alimentación y tierra y sus dimensiones son más reducidas (alojando por tanto menos nodos).

Además del conocimiento de la disposición eléctrica interna de una breadboard, es importante tener en cuenta una serie de consejos útiles para el día a día que nos vendrán bien a la hora de montar nuestros diseños. Para empezar, es recomendable utilizar siempre cable negro para las conexiones a tierra, cable rojo para alimentaciones de 5 V o más y cable verde (o cualquier otro color) para alimentaciones de 3 V (y así evitar dañar algún componente que no admita los 5 voltios). Estos colores son simplemente una convención generalizada (es decir, no son una norma establecida) pero es muy común seguirla por todo el mundo para evitar confusiones.

Otro consejo es observar si nuestra breadboard tiene pintadas las líneas roja y azul de sus buses de forma no continua. Si es el caso, significa que los puntos que forman cada bus no están conectados eléctricamente todos entre sí sino que hay un salto. Para empalmar todos los puntos de cada bus y así conseguir un único bus de alimentación y un único de bus de tierra (medida aconsejable para ganar claridad y comodidad en la realización de nuestros circuitos), lo que se debe hacer es unir mediante un cable un punto de cada extremo del salto para el bus de alimentación y unir mediante otro cable un punto de cada extremo del salto para el bus de tierra (lo que se llama realizar un “puente”).

Por otro lado, una precaución básica que hay que tener siempre en cuenta a la hora de utilizar el bus de tierra es la de procurar que todas las conexiones a tierra del circuito estén conectadas a su vez entre sí para que todo el circuito tenga la misma referencia (lo que a veces se llama “compartir las masas”). Es decir, que solamente exista una única tierra para todos los componentes. Esto es fundamental para que nuestros circuitos funcionen correctamente.

En el caso de protoboards que dispongan de los buses de alimentación y tierra en ambos lados (ya que existen breadboads con los buses disponibles solo en uno solo) podemos unir el bus de alimentación de un lado y del otro mediante un cable y el bus de tierra de un lado y de otro mediante otro cable. Esto nos servirá para que ambos pares de buses conduzcan corriente al conectar una fuente de alimentación, siendo así más fácil y ordenada la manipulación del circuito a montar.

Finalmente, recordar que es muy importante que a la hora de añadir, quitar o cambiar componentes en una breadboard esta no reciba alimentación eléctrica alguna. Si no se hace así, se corre el riesgo de recibir una descarga y/o dañar algún componente. También es importante comprobar que las partes metálicas de los cables (u otros componentes) no contacten entre sí porque esto provocaría un cortocircuito.

Por otro lado, además de las breadboards (protoboards), existen otros tipos diferentes de placas de prototipado, de los cuales las “perfboards” y las “stripboards” son los más importantes:

Perfboards: cumplen la misma función que las breadboards, pero consiguen que el prototipo del circuito sea más sólido. Constan básicamente de una placa rígida y delgada llena de agujeros pre-perforados ubicados en forma de cuadrícula y distanciados entre sí una distancia estándar. En estos agujeros debemos soldar nosotros los componentes de nuestro circuito y las uniones

entre éstos son realizadas con cables que hemos de soldar también a la placa. Más en concreto, los componentes se colocan encima de la cara superior de la placa (por lo que sus patillas atraviesan sus agujeros, siendo soldadas estas a la cara inferior) y los cables se sueldan por la cara inferior (permaneciendo por tanto relativamente ocultos en el montaje final del circuito). La cara inferior de una "perfboard" la podremos identificar fácilmente observando la presencia de anillos de cobre rodeando a los agujeros.

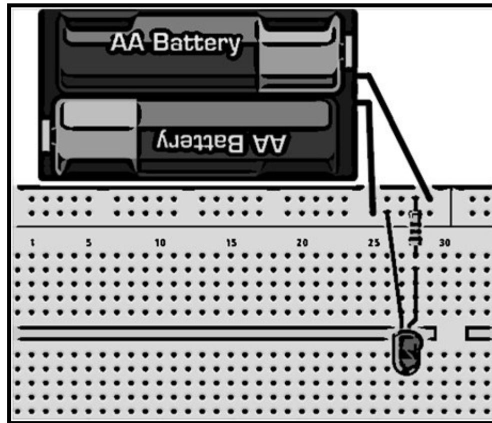
Stripboards: también conocidas con el nombre comercial registrado de "Veroboards"), son muy similares a las perfboards. La mayor diferencia está en que los agujeros de una perfboard están aislados eléctricamente entre sí (y por ello siempre se han de realizar las conexiones "manualmente") pero los agujeros de una cara de una stripboard están de entrada unidos por líneas de cobre conductor, en forma de filas paralelas independientes. En este sentido, las stripboards son similares a las breadboards, ya que ya vienen con una serie de nodos –conjunto de agujeros conectados entre sí– predefinidos. Las únicas soldaduras que hay que realizar por tanto son las de los propios componentes y las de los cables que conecten diferentes nodos.

Debido a que se requieren conocimientos de soldar (aunque sean mínimos) en los proyectos de este libro no usaremos ni perfboards ni stripboards. No obstante, si se quiere aprender a utilizar una "perfboard", un buen tutorial para ello es http://itp.nyu.edu/physcomp/Tutorials/SolderingAPerfBoard . Si se quiere aprender a utilizar una "stripboard", podemos consultar un buen tutorial en la dirección http://www.kpsec.freeuk.com/stripbd.htm.

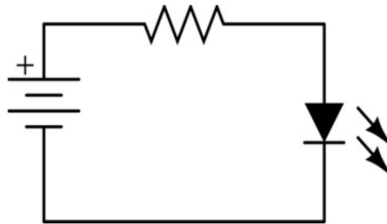
USO DE UNA PLACA DE PROTOTIPADO

En este apartado aparecen recopilados algunos ejemplos que muestran diferentes maneras de conectar dispositivos mediante una "breadboard".

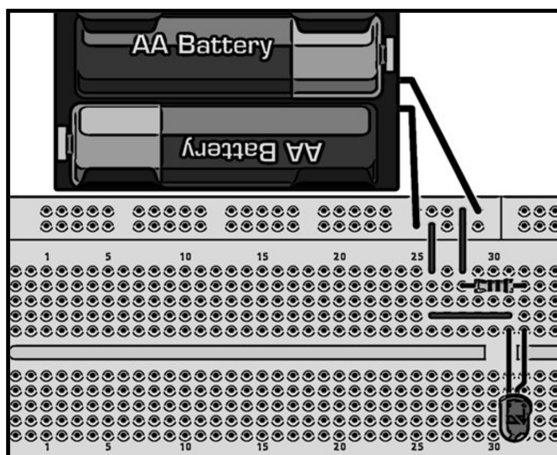
Ejemplo nº 1: en el siguiente diagrama se puede ver cómo se realiza un circuito donde una resistencia y un LED se conectan en serie.



En el dibujo anterior el polo positivo de la fuente de alimentación se conecta al bus más exterior. Esto hará que cualquier elemento conectado a algún agujero de ese bus reciba directamente de allí la alimentación eléctrica. Eso es lo que le pasa precisamente a la resistencia de nuestro circuito: su terminal superior está enchufado en ese bus. El otro terminal está colocado en un nodo de la breadboard, nodo donde precisamente se conecta también el terminal positivo del LED (en el diagrama, es el que aparece con una pequeña hendidura en su raíz) . Esto quiere decir que estos dos componentes están directamente conectados. Finalmente, el terminal negativo del LED está enchufado en el bus más interior de la breadboard, bus en el cual se conecta también el polo negativo de la fuente, por lo que el LED está directamente conectado a él, cerrando el círculo. El circuito anterior tiene un esquema como este:

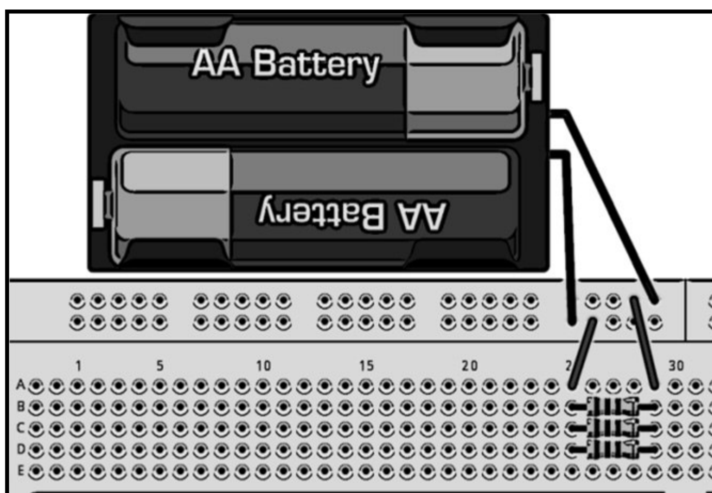


También se podría haber hecho el mismo circuito llevando el cable de alimentación y el de tierra a la zona de nodos:

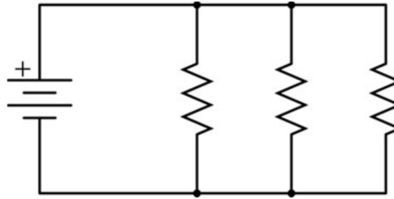


Es importante tener en cuenta que todos los agujeros de un mismo nodo representan un único punto de conexión (un error muy común al principio es conectar ambos terminales de un dispositivo en un mismo nodo, cosa que no tiene sentido). Sabiendo esto, es fácil ver en la ilustración anterior que el terminal izquierdo de la resistencia está conectado a un cable que a su vez está conectado a la alimentación, y que el terminal derecho de la resistencia está conectado al terminal positivo del LED mientras que su terminal negativo está conectado a un cable que a su vez está conectado a otro, el cual va a parar finalmente a tierra.

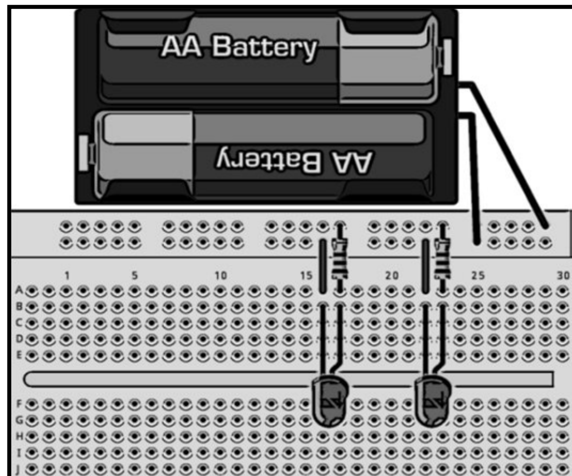
Ejemplo nº 2: La siguiente ilustración muestra la conexión de tres dispositivos en paralelo (concretamente, tres resistencias):



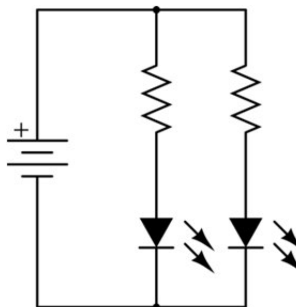
Su esquema equivalente sería:



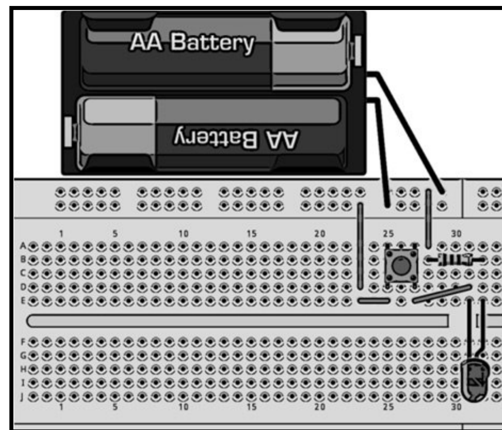
Ejemplo nº 3: El siguiente esquema muestra la conexión en serie de una resistencia y un LED, y la conexión en paralelo de ambos a otra resistencia y LED. Si probamos este circuito en la realidad, veremos que dependiendo del valor de cada resistencia, el LED correspondiente se iluminará más o menos.



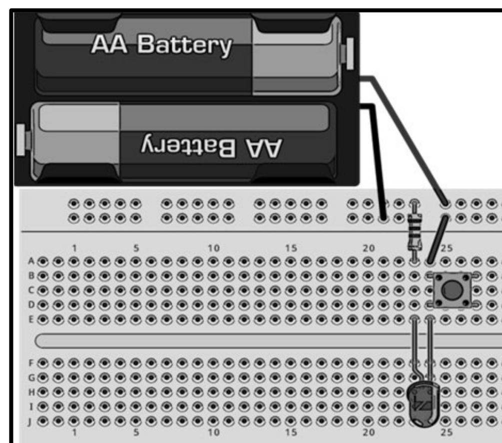
El circuito anterior tiene un esquema como este:



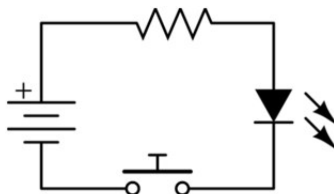
Ejemplo nº 4: El siguiente dibujo muestra la conexión de tres dispositivos en serie: un LED, una resistencia y un pulsador:



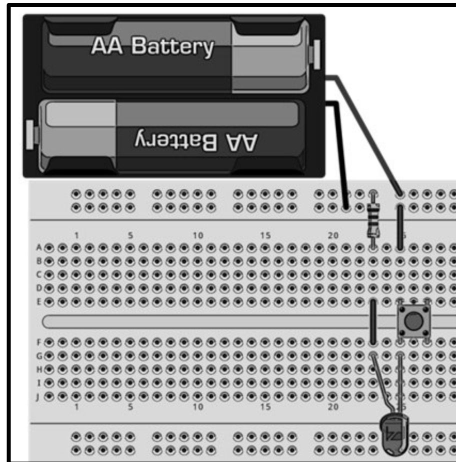
También se podría haber realizado el mismo circuito sin tanto cable:



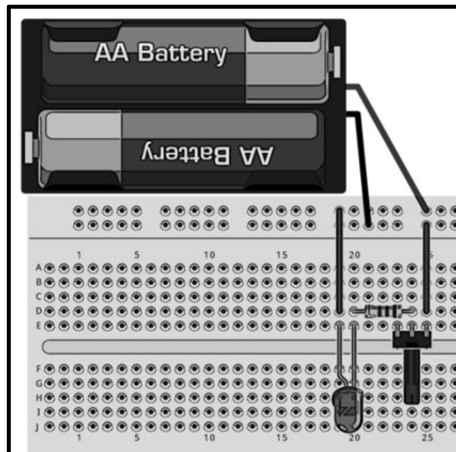
En cualquier caso, el esquema equivalente del circuito anterior sería:



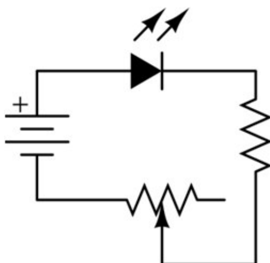
Atención: fijarse en las figuras anteriores cómo están ubicadas las patillas del pulsador. Si en el último circuito hubiéramos colocado el pulsador de la manera que se muestra en la siguiente figura, el LED estaría encendido siempre porque, independientemente del estado del pulsador, las dos patillas enfrentadas siempre están unidas internamente (ya que en realidad, representan un único punto de conexión, tal como ya hemos estudiado).



Ejemplo nº 5: El siguiente esquema muestra la conexión de tres dispositivos en serie: un LED, una resistencia y un potenciómetro:



Su esquema correspondiente es este (donde podemos ver claramente que se conecta la patilla central del potenciómetro y uno de sus extremos, pero no el otro):



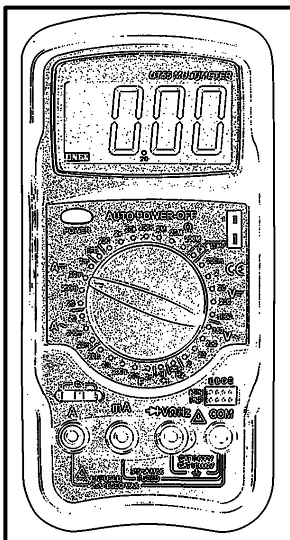
Tenemos que imaginar que la flecha central del símbolo del potenciómetro se moverá desde un extremo hasta el otro de ese símbolo según giremos la rosca del potenciómetro. Tal como está dibujado el esquema anterior, si la flecha se "sitúa" en el extremo derecho del símbolo, el potenciómetro funcionará con su valor de resistencia máximo; si la flecha se "sitúa" en el extremo izquierdo, el potenciómetro no ofrecerá resistencia alguna.

Este último hecho es la causa de haber añadido una resistencia entre el LED y el potenciómetro: si en algún momento ajustáramos el potenciómetro a cero, ¡no habría ninguna resistencia en el circuito!, y esto dañaría irreversiblemente el LED porque recibiría demasiada intensidad de corriente. Por tanto, la resistencia adicional mantiene un valor mínimo que no es rebajado nunca.

Finalmente, y por otro lado, si se desea alimentar un circuito montado sobre una breadboard mediante un adaptador AC/DC en vez de con pilas/baterías (tal como se ha mostrado en los ejemplos anteriores), podemos hacer uso de unas pequeñas plaquitas especialmente pensadas para ser conectadas por un lado a la clavija de 5,5/2,1 mm de dicho adaptador (gracias al zócalo pertinente que incorporan) y por otro a la breadboard. Además, estas plaquitas contienen un regulador de voltaje propio que permite adaptar la tensión recibida del adaptador a tensiones rebajadas y estables (generalmente de 5 V o 3,3 V), más adecuadas para el tipo de circuitos que generalmente se montan en una breadboard. Ejemplos de estas plaquitas (de entre muchas otras disponibles en los distintos distribuidores mencionados en el apéndice A) son el producto nº 184 de Adafruit, el nº 114 (o también el nº 10804) de Sparkfun, el llamado "5 V/3,3 V Breadboard DC-DC Power Supply" (y también el "5V Breadboard USB Power Supply", el cual incluye además un zócalo USB mini-B para recibir alimentación también por allí) de Akafugu o el "Breadboard Power Supply Module" de IteadStudio (también con zócalo USB mini-B). En todo caso, se recomienda consultar la documentación oficial de cada producto para conocer de forma exhaustiva sus funcionalidades y limitaciones.

USO DE UN MULTÍMETRO DIGITAL

El multímetro digital es un instrumento que sirve para medir alguna de las tres magnitudes relacionadas por la Ley de Ohm: o bien el voltaje existente entre dos puntos de un circuito, o bien la intensidad de corriente que fluye a través de él, o bien la resistencia que ofrece cierto componente. Dependiendo del modelo, también hay multímetros que pueden medir otras magnitudes como la capacidad de los condensadores, y más. Es decir, es una herramienta que nos permite comprobar el correcto funcionamiento de los componentes y circuitos electrónicos, por lo que es fundamental tenerla a mano cuando realicemos nuestros proyectos.



Existen muchos modelos diferentes de multímetros digitales, por lo que es importante leer el manual de instrucciones del fabricante para asegurar el buen funcionamiento del instrumento. Un ejemplo puede ser el producto nº 9141 de Sparkfun. De todas formas, aunque dependiendo del modelo puedan cambiar la posición de sus elementos y la cantidad de funciones, en general podemos identificar las partes y funciones estándar de un multímetro genérico como las siguientes:

Botón de “power” (apagado-encendido): la mayoría de multímetros son alimentados mediante pilas.

Display: pantalla de cristal líquido en donde se mostrarán los resultados de las mediciones.

Llave selectora: sirve para elegir el tipo de magnitud a medir y el rango de medición. Los símbolos que la rodean indican el tipo de magnitud a medir, y los más comunes son el voltaje directo (V-) y alterno (V~), la corriente directa (A-) y alterna (A~), la resistencia (Ω), la capacidad (F) o la frecuencia (Hz). Los números que rodean la llave indican el rango de medición. Para entender esto último, supongamos que los números posibles para el voltaje continuo son por ejemplo “200 mV” ,”2 V”,”20 V” y “200 V”; esto querrá decir que en la posición “200 mV” se podrán medir voltajes desde 0 hasta este valor como máximo; en la posición “2 V” se podrán medir voltajes superiores a 200mV pero inferiores a 2 V; en la posición “20 V” se podrán medir voltajes superiores a 2 V pero inferiores 20 V, y así, mostrándose en el display los valores numéricos medidos de acuerdo a la escala elegida.

Cables rojo y negro con punta: el cable negro siempre se conectará al zócalo negro del multímetro (solo existe uno, y generalmente está señalado con la palabra “COM” –de “referencia COMún” –), mientras que el cable rojo se conectará al zócalo rojo adecuado según la magnitud que se quiera medir (ya que hay varios): si se quiere medir voltaje, resistencia o frecuencia (tanto en continua como en alterna), se deberá conectar el cable rojo al zócalo rojo marcado normalmente con el símbolo “+VΩHz” ; si se quiere medir intensidad de corriente (tanto en continua como en alterna), se deberá conectar el cable rojo al zócalo rojo marcado con el símbolo “mA” o bien “A”, dependiendo del rango a medir.

Una vez conocidas las partes funcionales de esta herramienta, la podemos utilizar para realizar diferentes medidas:

Para medir el voltaje (continuo) existente entre dos puntos de un circuito alimentado, deberemos conectar los cables convenientemente al multímetro para colocar seguidamente la punta del cable negro en un punto del circuito y la del cable rojo en el otro (de tal forma que en realidad estemos realizando una conexión en paralelo con dicho circuito). Seguidamente, moveremos la llave selectora al símbolo V- y elegiremos el rango de medición adecuado. Si este lo desconocemos, lo que podemos hacer es empezar por el rango más elevado e ir bajando paso a paso para obtener finalmente la precisión deseada. Si bajamos más de la cuenta (es decir, si el valor a medir es mayor que el rango elegido), lo sabremos porque a la izquierda del display se mostrará el valor especial “1”.

También podemos utilizar la posibilidad ofrecida por el multímetro de medir voltaje continuo para conocer la diferencia de potencial generada por una determinada fuente de alimentación (y así saber en el caso de una pila, por ejemplo, si está gastada o no). En este caso, deberíamos colocar la punta del cable rojo en el borne positivo de la pila y el negro en el negativo y proceder de la misma manera, seleccionando la magnitud y rango a medir.

Para medir la resistencia de un componente, debemos mantener desconectado dicho componente para que no reciba corriente de ningún circuito. El procedimiento para medir una resistencia es bastante similar al de medir tensiones: basta con conectar cada terminal del componente a los cables del multímetro (si el componente tiene polaridad, como es el caso de los diodos y de algunos condensadores, el cable rojo se ha de conectar al terminal positivo del componente y el negro al negativo; si el componente no

tiene polaridad, esto es indiferente) y colocar el selector en la posición de ohmios y en la escala apropiada al tamaño de la resistencia que se desea medir. Si no sabemos aproximadamente el rango de la resistencia a medir, empezaremos colocando la ruleta en la escala más grande, e iremos reduciendo la escala hasta que encontremos la que más precisión nos dé sin salirnos de rango. Igualmente, si la escala elegida resulta ser menor que el valor a medir, el display indicará “1” a su izquierda; en ese caso, por tanto, habrá que ir subiendo de rango hasta encontrar el correcto.

Para medir la intensidad que fluye por un circuito, hay que conectar el multímetro en serie con el circuito en cuestión. Por eso, para medir intensidades tendremos que abrir el circuito para intercalar el multímetro en medio, con el propósito de que la intensidad circule por su interior. Concretamente, el proceso a seguir es: insertar el cable rojo en el zócalo adecuado (mA o A según la cantidad de corriente a medir) y el cable negro en el zócalo negro, empalmar cada cable del multímetro en cada uno de los dos extremos del circuito abierto que tengamos (cerrándolo así, por lo tanto) y ajustar el selector a la magnitud y rango adecuados.

Idealmente, el multímetro funcionando como medidor de corriente tiene una resistencia nula al paso de la corriente a través de él (precisamente para evitar alteraciones en la medida del valor de la intensidad real), por lo que está relativamente desprotegido de intensidades muy elevadas y pueda dañarse con facilidad. Hay que tener siempre en cuenta por tanto el máximo de corriente que puede soportar, el cual lo ha de indicar el fabricante (además del tiempo máximo que puede estar funcionando en este modo).

Para medir la capacidad de un condensador, también podemos utilizar la mayoría de multímetros digitales del mercado. Tan solo tendremos que conectar las patillas del condensador a unos zócalos especiales para ello, marcados con la marca ”CX”. Los condensadores deben estar descargados antes de conectarlos a dichos zócalos. Para los condensadores que tengan polaridad habrá que identificar el zócalo correspondiente a cada polo en el manual del fabricante.

Para medir continuidad (es decir para comprobar si dos puntos de un circuito están eléctricamente conectados), simplemente se debe ajustar el selector en la posición marcada con el signo de una “onda de audio” y conectar los dos cables a cada punto a medir (no importa la polaridad). Atención: este modo solo se puede utilizar cuando el circuito a medir <u>no</u> está recibiendo

alimentación eléctrica. Si hay continuidad, el multímetro emitirá un sonido (gracias a un zumbador que lleva incorporado); si no, no se escuchará nada. También se puede observar lo que muestra el display según el caso, pero el mensaje concreto depende del modelo, así que se recomienda consultar las instrucciones de cada aparato en particular.

Una aplicación práctica del multímetro funcionando como medidor de continuidad es la comprobación de qué agujeros de una breadboard pertenecientes al mismo nodo mantienen su conectividad, ya que después de un uso continuado es relativamente fácil que esta se estropee.

2 HARDWARE ARDUINO

¿QUÉ ES UN SISTEMA ELECTRÓNICO?

Un sistema electrónico es un conjunto de: sensores, circuitería de procesamiento y control, actuadores y fuente de alimentación.

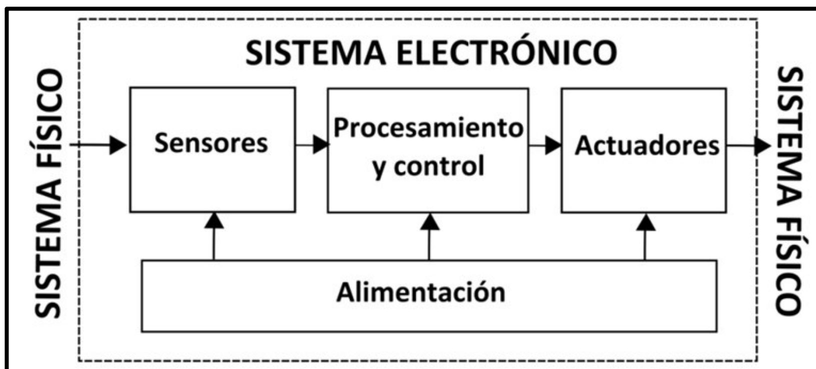
Los sensores obtienen información del mundo físico externo y la transforman en una señal eléctrica que puede ser manipulada por la circuitería interna de control. Existen sensores de todo tipo: de temperatura, de humedad, de movimiento, de sonido (micrófonos), etc.

Los circuitos internos de un sistema electrónico procesan la señal eléctrica convenientemente. La manipulación de dicha señal dependerá tanto del diseño de los diferentes componentes hardware del sistema, como del conjunto lógico de instrucciones (es decir, del “programa”) que dicho hardware tenga pregrabado y que sea capaz de ejecutar de forma autónoma.

Los actuadores transforman la señal eléctrica acabada de procesar por la circuitería interna en energía que actúa directamente sobre el mundo físico externo. Ejemplos de actuadores son: un motor (energía mecánica), una bombilla (energía lumínica), un altavoz (energía acústica), etc.

La fuente de alimentación proporciona la energía necesaria para que se pueda realizar todo el proceso descrito de “obtención de información del medio <->

procesamiento <-> actuación sobre el medio". Ejemplos de fuentes son las pilas, baterías, adaptadores AC/DC, etc.



¿QUÉ ES UN MICROCONTROLADOR?

Un microcontrolador es un circuito integrado o "chip" (es decir, un dispositivo electrónico que integra en un solo encapsulado un gran número de componentes) que tiene la característica de ser programable. Es decir, que es capaz de ejecutar de forma autónoma una serie de instrucciones previamente definidas por nosotros. En el diagrama anterior, representativo de un sistema electrónico, el microcontrolador sería el componente principal de la circuitería de procesamiento y control.

Por definición, un microcontrolador (también llamado comúnmente "micro") ha de incluir en su interior tres elementos básicos:

CPU (Unidad Central de Proceso): es la parte encargada de ejecutar cada instrucción y de controlar que dicha ejecución se realice correctamente. Normalmente, estas instrucciones hacen uso de datos disponibles previamente (los "datos de entrada"), y generan como resultado otros datos diferentes (los "datos de salida"), que podrán ser utilizados (o no) por la siguiente instrucción.

Diferentes tipos de memorias: son en general las encargadas de alojar tanto las instrucciones como los diferentes datos que estas necesitan. De esta manera posibilitan que toda esta información (instrucciones y datos) esté siempre disponible para que la CPU pueda acceder y trabajar con ella en

cualquier momento. Generalmente encontraremos dos tipos de memorias: las que su contenido se almacena de forma permanente incluso tras cortes de alimentación eléctrica (llamadas "persistentes"), y las que su contenido se pierde al dejar de recibir alimentación (llamadas "volátiles"). Según las características de la información a guardar, esta se grabará en un tipo u otro de memoria de forma automática, habitualmente.

Diferentes patillas de E/S (entrada/salida): son las encargadas de comunicar el microcontrolador con el exterior. En las patillas de entrada del microcontrolador podremos conectar sensores para que este pueda recibir datos provenientes de su entorno, y en sus patillas de salida podremos conectar actuadores para que el microcontrolador pueda enviarles órdenes e así interactuar con el medio físico. De todas formas, muchas patillas de la mayoría de microcontroladores no son exclusivamente de entrada o de salida, sino que pueden ser utilizados indistintamente para ambos propósitos (de ahí el nombre de E/S).

Es decir, un microcontrolador es un computador completo (aunque con prestaciones limitadas) en un solo chip, el cual está especializado en ejecutar constantemente un conjunto de instrucciones predefinidas. Estas instrucciones irán teniendo en cuenta en cada momento la información obtenida y enviada por las patillas de E/S y reaccionarán en consecuencia. Lógicamente, las instrucciones serán diferentes según el uso que se le quiera dar al microcontrolador, y deberemos de decidir nosotros cuáles son.

Cada vez existen más productos domésticos que incorporan algún tipo de microcontrolador con el fin de aumentar sustancialmente sus prestaciones, reducir su tamaño y coste, mejorar su fiabilidad y disminuir el consumo. Así, podemos encontrar microcontroladores dentro de multitud de dispositivos electrónicos que usamos en nuestra vida diaria, como pueden ser desde un simple timbre hasta un completo robot pasando por juguetes, frigoríficos, televisores, lavadoras, microondas, impresoras, el sistema de arranque de nuestro coche, etc.

¿QUÉ ES ARDUINO?

Arduino (http://www.arduino.cc) es en realidad tres cosas:

Una placa hardware libre que incorpora un microcontrolador reprogramable y una serie de pines-hembra (los cuales están unidos internamente a las

patillas de E/S del microcontrolador) que permiten conectar allí de forma muy sencilla y cómoda diferentes sensores y actuadores.

Cuando hablamos de “placa hardware” nos estamos refiriendo en concreto a una PCB (del inglés “printed circuit board”, o sea, placa de circuito impreso). Las PCBs son superficies fabricadas de un material no conductor (normalmente resinas de fibra de vidrio reforzada, cerámica o plástico) sobre las cuales aparecen laminadas (“pegadas”) pistas de material conductor (normalmente cobre). Las PCBs se utilizan para conectar eléctricamente, a través de los caminos conductores, diferentes componentes electrónicos soldados a ella. Una PCB es la forma más compacta y estable de construir un circuito electrónico (en contraposición a una breadboard, perfboard o similar) pero, al contrario que estas, una vez fabricada, su diseño es bastante difícil de modificar. Así pues, la placa Arduino no es más que una PCB que implementa un determinado diseño de circuitería interna.

No obstante, cuando hablamos de “placa Arduino”, deberíamos especificar el modelo concreto, ya que existen varias placas Arduino oficiales, cada una con diferentes características (como el tamaño físico, el número de pines-hembra ofrecidos, el modelo de microcontrolador incorporado –y como consecuencia, entre otras cosas, la cantidad de memoria utilizable–, etc.). Conviene conocer estas características para identificar qué placa Arduino es la que nos convendrá más en cada proyecto.

De todas formas, aunque puedan ser modelos específicos diferentes (tal como acabamos de comentar), los microcontroladores incorporados en las diferentes placas Arduino pertenecen todos a la misma “familia tecnológica”, por lo que su funcionamiento en realidad es bastante parecido entre sí. En concreto, todos los microcontroladores son de tipo AVR, una arquitectura de microcontroladores desarrollada y fabricada por la marca Atmel (http://www.atmel.com). Es por eso que, en este libro seguiremos nombrando “placa Arduino” a cualquiera de ellas mientras no sea imprescindible hacer algún tipo de distinción.

El diseño hardware de la placa Arduino está inspirado originalmente en el de otra placa de hardware libre preexistente, la placa Wiring (http://www.wiring.co). Esta placa surgió en 2003 como proyecto personal de Hernando Barragán, estudiante por aquel entonces del Instituto de Diseño de Ivrea (lugar donde surgió en 2005 precisamente la placa Arduino).

Un software (más en concreto, un “entorno de desarrollo”) **gratis, libre y multiplataforma** (ya que funciona en Linux, MacOS y Windows) que debemos

instalar en nuestro ordenador y que nos permite escribir, verificar y guardar ("cargar") en la memoria del microcontrolador de la placa Arduino el conjunto de instrucciones que deseamos que este empiece a ejecutar. Es decir: nos permite programarlo. La manera estándar de conectar nuestro computador con la placa Arduino para poder enviarle y grabarle dichas instrucciones es mediante un simple cable USB, gracias a que la mayoría de placas Arduino incorporan un conector de este tipo.

Los proyectos Arduino pueden ser autónomos o no. En el primer caso, una vez programado su microcontrolador, la placa no necesita estar conectada a ningún computador y puede funcionar autónomamente si dispone de alguna fuente de alimentación. En el segundo caso, la placa debe estar conectada de alguna forma permanente (por cable USB, por cable de red Ethernet, etc.) a un computador ejecutando algún software específico que permita la comunicación entre este y la placa y el intercambio de datos entre ambos dispositivos. Este software específico lo deberemos programar generalmente nosotros mismos mediante algún lenguaje de programación estándar como Python, C, Java, Php, etc., y será independiente completamente del entorno de desarrollo Arduino, el cual no se necesitará más, una vez que la placa ya haya sido programada y esté en funcionamiento.

Un lenguaje de programación libre. Por "lenguaje de programación" se entiende cualquier idioma artificial diseñado para expresar instrucciones (siguiendo unas determinadas reglas sintácticas) que pueden ser llevadas a cabo por máquinas. Concretamente dentro del lenguaje Arduino, encontramos elementos parecidos a muchos otros lenguajes de programación existentes (como los bloques condicionales, los bloques repetitivos, las variables, etc.), así como también diferentes comandos –asimismo llamados "órdenes" o "funciones" – que nos permiten especificar de una forma coherente y sin errores las instrucciones exactas que queremos programar en el microcontrolador de la placa. Estos comandos los escribimos mediante el entorno de desarrollo Arduino.

Tanto el entorno de desarrollo como el lenguaje de programación Arduino están inspirado en otro entorno y lenguaje libre preexistente: Processing (http://www.processing.org), desarrollado inicialmente por Ben Fry y Casey Reas. Que el software Arduino se parezca tanto a Processing no es casualidad, ya que este está especializado en facilitar la generación de imágenes en tiempo real, de animaciones y de interacciones visuales, por lo que muchos profesores del Instituto de Diseño de Ivrea lo utilizaban en sus clases. Como fue en ese centro donde precisamente se inventó Arduino es natural que ambos entornos y lenguajes guarden

bastante similitud. No obstante, hay que aclarar que el lenguaje Processing está construido internamente con código escrito en lenguaje Java, mientras que el lenguaje Arduino se basa internamente en código C/C++.

Con Arduino se pueden realizar multitud de proyectos de rango muy variado: desde robótica hasta domótica, pasando por monitorización de sensores ambientales, sistemas de navegación, telemática, etc. Realmente, las posibilidades de esta plataforma para el desarrollo de productos electrónicos son prácticamente infinitas y tan solo están limitadas por nuestra imaginación.

¿CUÁL ES EL ORIGEN DE ARDUINO?

Arduino nació en el año 2005 en el Instituto de Diseño Interactivo de Ivrea (Italia), centro académico donde los estudiantes se dedicaban a experimentar con la interacción entre humanos y diferentes dispositivos (muchos de ellos basados en microcontroladores) para conseguir generar espacios únicos, especialmente artísticos. Arduino apareció por la necesidad de contar con un dispositivo para utilizar en las aulas que fuera de bajo coste, que funcionase bajo cualquier sistema operativo y que contase con documentación adaptada a gente que quisiera empezar de cero. La idea original fue, pues, fabricar la placa para uso interno de la escuela.

No obstante, el Instituto se vio obligado a cerrar sus puertas precisamente en 2005. Ante la perspectiva de perder en el olvido todo el desarrollo del proyecto Arduino que se había ido llevando a cabo durante aquel tiempo, se decidió liberarlo y abrirlo a "la comunidad" para que todo el mundo tuviera la posibilidad de participar en la evolución del proyecto, proponer mejoras y sugerencias y mantenerlo "vivo". Y así ha sido: la colaboración de muchísima gente ha hecho que Arduino poco a poco haya llegado a ser lo que es actualmente: un proyecto de hardware y software libre de ámbito mundial.

El principal responsable de la idea y diseño de Arduino, y la cabeza visible del proyecto es el llamado "Arduino Team", formado por Massimo Banzi (profesor en aquella época del Instituto Ivrea), David Cuartielles (profesor de la Escuela de Artes y Comunicación de la Universidad de Mälmo, Suecia), David Mellis (por aquel entonces estudiante en Ivrea y actualmente miembro del grupo de investigación High-Low Tech del MIT Media Lab), Tom Igoe (profesor de la Escuela de Arte Tisch de Nueva York), y Gianluca Martino (responsable de empresa fabricante de los prototipos de las placas, cuya web oficial es: http://www.smartprojects.it).

Existe un documental de 30 minutos muy interesante, en el cual interviene el "Arduino Team" explicando en primera persona todo el proceso de gestación y evolución del proyecto Arduino, desde los detalles técnicos que se tuvieron en cuenta hasta la filosofía libre que impregnó (e impregna) su desarrollo, pasando por diferentes testimonios de colaboradores de todo el mundo. Se puede ver gratuitamente en http://arduinothedocumentary.org.

¿QUÉ QUIERE DECIR QUE ARDUINO SEA "SOFTWARE LIBRE"?

En párrafos anteriores hemos comentado que Arduino es una placa de "hardware *libre*" y también "un entorno y lenguaje de programación (es decir, software) *libre*". ¿Pero qué significa aquí la palabra "libre" exactamente?

Según la Free Software Foundation (http://www.fsf.org), organización encargada de fomentar el uso y desarrollo del software libre a nivel mundial, un software para ser considerado libre ha de ofrecer a cualquier persona u organización cuatro libertades básicas e imprescindibles:

Libertad 0: la libertad de usar el programa con cualquier propósito y en cualquier sistema informático.

Libertad 1: la libertad de estudiar cómo funciona internamente el programa, y adaptarlo a las necesidades particulares. El acceso al código fuente es un requisito previo para esto.

Libertad 2: la libertad de distribuir copias.

Libertad 3: la libertad de mejorar el programa y hacer públicas las mejoras a los demás, de modo que toda la comunidad se beneficie. El acceso al código fuente es un requisito previo para esto.

Un programa es software libre si los usuarios tienen todas estas libertades. Así pues, el software libre es aquel software que da a los usuarios la libertad de poder ejecutarlo, copiarlo y distribuirlo (a cualquiera y a cualquier lugar), estudiarlo, cambiarlo y mejorarlo, sin tener que pedir ni pagar permisos al desarrollador original ni a ninguna otra entidad específica. La distribución de las copias puede ser con o sin modificaciones propias, y atención, puede ser gratis ¡o no!: el "software libre" es un asunto de libertad, no de precio.

Para que un programa sea considerado libre a efectos legales ha de someterse a algún tipo de licencia de distribución, entre las cuales se encuentran la licencia GPL (General Public License), o la LGPL, entre otras. El tema de las diferentes licencias es un poco complicado: hay muchas y con muchas cláusulas. Para saber más sobre este tema, se puede consultar http://www.opensource.org/licenses/category , donde está disponible el texto oficial original de las licencias más importantes. Ejemplos de software libre hay muchos: el kernel Linux, el navegador Firefox, la suite ofimática LibreOffice, el reproductor multimedia VLC, etc.

El software Arduino es software libre porque se publica con una combinación de la licencia GPL (para el entorno visual de programación propiamente dicho) y la licencia LGPL (para los códigos fuente de gestión y control del microcontrolador a nivel más interno). La consecuencia de esto es, en pocas palabras, que cualquier persona que quiera (y sepa), puede formar parte del desarrollo del software Arduino y contribuir así a mejorar dicho software, aportando nuevas características, sugiriendo ideas de nuevas funcionalidades, compartiendo soluciones a posibles errores existentes, etc. Esta manera de funcionar provoca la creación espontánea de una comunidad de personas que colaboran mutuamente a través de Internet, y consigue que el software Arduino evolucione según lo que la propia comunidad decida. Esto va mucho más allá de la simple cuestión de si el software Arduino es gratis o no, porque el usuario deja de ser un sujeto pasivo para pasar a ser (si quiere) un sujeto activo y partícipe del proyecto.

¿QUÉ QUIERE DECIR QUE ARDUINO SEA “HARDWARE LIBRE”?

El hardware libre (también llamado “open-source” o “de fuente abierta”) comparte muchos de los principios y metodologías del software libre. En particular, el hardware libre permite que la gente pueda estudiarlo para entender su funcionamiento, modificarlo, reutilizarlo, mejorarlo y compartir dichos cambios. Para conseguir esto, la comunidad ha de poder tener acceso a los ficheros esquemáticos del diseño del hardware en cuestión (que son ficheros de tipo CAD). Estos ficheros detallan toda la información necesaria para que cualquier persona con los materiales, herramientas y conocimientos adecuados pueda reconstruir dicho hardware por su cuenta sin problemas, ya que consultando estos ficheros se puede conocer qué componentes individuales integran el hardware y qué interconexiones existen entre cada uno de ellos.

La placa Arduino es hardware libre porque sus ficheros esquemáticos están disponibles para descargar de la página web del proyecto con la licencia Creative

Commons Attribution Share-Alike (http://es.creativecommons.org/licencia), la cual es una licencia libre que permite realizar trabajos derivados tanto personales como comerciales (siempre que estos den crédito a Arduino y publiquen sus diseños bajo la misma licencia). Así pues, uno mismo se puede construir su propia placa Arduino "a mano". No obstante, lo más normal es comprarlas de un distribuidor ya preensambladas y listas para usar; en ese caso, lógicamente, la placa Arduino, aunque sea libre, no puede ser gratuita, ya que es un objeto físico y su fabricación cuesta dinero.

A diferencia del mundo del software libre, donde el ecosistema de licencias libres es muy rico y variado, en el ámbito del hardware todavía no existen prácticamente licencias específicamente de hardware libre, ya que el concepto de "hardware libre" es relativamente nuevo. De hecho, hasta hace poco no existía un consenso generalizado en su definición. Para empezar a remediar esta situación, en el año 2010 surgió el proyecto OSHD (http://freedomdefined.org/OSHW), el cual pretende establecer una colección de principios que ayuden a identificar como "hardware libre" un producto físico. OSHD no es una licencia (es decir, un contrato legal), sino una declaración de intenciones (es decir, una lista general de normas y de características) aplicable a cualquier artefacto físico para que pueda ser considerado libre. El objetivo de la OSHD (en cuya redacción ha participado gente relacionada con el proyecto Arduino, entre otros) es ofrecer un marco de referencia donde se respete por un lado la libertad de los creadores para controlar su propia tecnología y al mismo tiempo se establezcan los mecanismos adecuados para compartir el conocimiento y fomentar el comercio a través del intercambio abierto de diseños. En otras palabras: mostrar que puede existir una alternativa a las patentes de hardware que tampoco sea necesariamente el dominio público. El proyecto OSHD abre, pues, un camino que crea precedentes legales para facilitar el siguiente paso lógico del proceso: la creación de licencias libres de hardware.

El objetivo del hardware libre es, por lo tanto, facilitar y acercar la electrónica, la robótica y en definitiva la tecnología actual a la gente, no de una manera pasiva, meramente consumista, sino de manera activa, involucrando al usuario final para que entienda y obtenga más valor de la tecnología actual e incluso ofreciéndole la posibilidad de participar en la creación de futuras tecnologías. Básicamente, el hardware abierto significa tener la posibilidad de mirar qué es lo que hay dentro de las cosas, y que eso sea éticamente correcto. Permite, en definitiva, mejorar la educación de las personas. Por eso el concepto de software y hardware libre es tan importante, no solo para el mundo de la informática y de la electrónica, sino para la vida en general.

¿POR QUÉ ELEGIR ARDUINO?

Existen muchas otras placas de diferentes fabricantes que, aunque incorporan diferentes modelos de microcontroladores, son comparables y ofrecen una funcionalidad más o menos similar a la de las placas Arduino. Todas ellas también vienen acompañadas de un entorno de desarrollo agradable y cómodo y de un lenguaje de programación sencillo y completo. No obstante, la plataforma Arduino (hardware + software) ofrece una serie de ventajas:

Arduino es libre y extensible: esto quiere decir que cualquiera que desee ampliar y mejorar tanto el diseño hardware de las placas como el entorno de desarrollo software y el propio lenguaje de programación, puede hacerlo sin problemas. Esto permite que exista un rico "ecosistema" de extensiones, tanto de variantes de placas no oficiales como de librerías software de terceros, que pueden adaptarse mejor a nuestras necesidades concretas.

Breve nota sobre las librerías:

Una librería (mala traducción del inglés "library", que significa "biblioteca") es un conjunto de instrucciones de un lenguaje de programación agrupadas de una forma coherente. Podemos entender una librería como un "cajón" etiquetado que guarda unas determinadas instrucciones relacionadas entre sí. Así, tenemos el "cajón" de las instrucciones para controlar motores, el "cajón" de las instrucciones para almacenar datos en una tarjeta de memoria, etc.

Las librerías sirven para proveer funcionalidad extra (manipulando datos, interactuando con hardware, etc.) pero también para facilitar el desarrollo de nuestros proyectos, porque están diseñadas para que a la hora de escribir nuestro programa no tengamos que hacer "la faena sucia" de conocer todos los detalles técnicos sobre el manejo de un determinado hardware, o ser un experto en la configuración de determinado componente, ya que las librerías ocultan esa complejidad.

Además, la organización del lenguaje en librerías permite que los programas resulten más pequeños y sean escritos de una forma más flexible y modular, ya que solo están disponibles en cada momento las instrucciones ofrecidas por las librerías utilizadas en ese instante.

El lenguaje Arduino incorpora por defecto una serie de librerías oficiales que iremos estudiando a lo largo de este libro (son las que aparecen listadas aquí:

http://arduino.cc/en/Reference/Libraries), pero también ofrece la posibilidad de utilizar librerías creadas por terceros (hay literalmente decenas) que amplían la funcionalidad del propio lenguaje y permiten que la placa Arduino se adapte a multitud de escenarios diferentes.

Arduino tiene una gran comunidad: muchas personas lo utilizan, enriquecen la documentación y comparten continuamente sus ideas.

Su entorno de programación es multiplataforma: se puede instalar y ejecutar en sistemas Windows, Mac OS X y Linux. Esto no ocurre con el software de muchas otras placas.

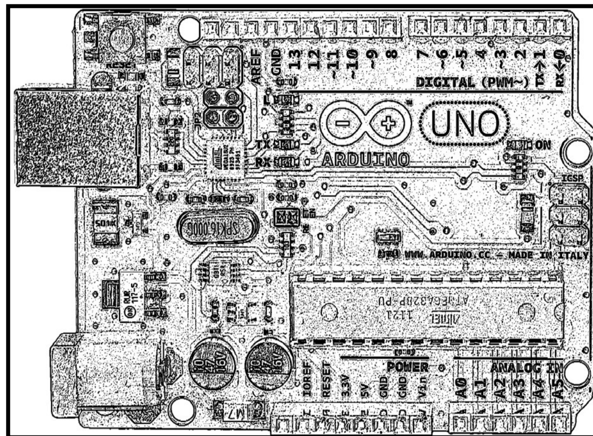
Su entorno y el lenguaje de programación son simples y claros: son muy fáciles de aprender y de utilizar, a la vez que flexibles y completos para que los usuarios avanzados puedan aprovechar y exprimir todas las posibilidades del hardware. Además, están bien documentados, con ejemplos detallados y gran cantidad de proyectos publicados en diferentes formatos.

Las placas Arduino son baratas: la placa Arduino estándar (llamada Arduino UNO) ya preensamblada y lista para funcionar cuesta alrededor de 20 euros. Incluso, uno mismo se la podría construir (Arduino es hardware libre, recordemos) adquiriendo los componentes por separado, con lo que el precio total de la placa resultante sería incluso menor.

Las placas Arduino son reutilizables y versátiles: reutilizables porque se puede aprovechar la misma placa para varios proyectos (ya que es muy fácil de desconectarla, reconectarla y reprogramarla), y versátiles porque las placas Arduino proveen varios tipos diferentes de entradas y salidas de datos, los cuales permiten capturar información de sensores y enviar señales a actuadores de múltiples formas.

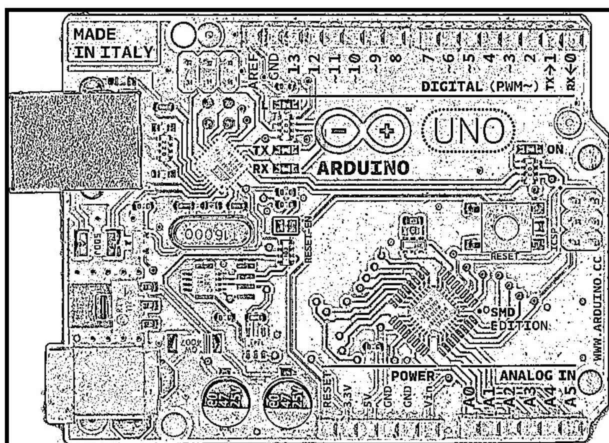
CARACTERÍSTICAS DEL MICRO DE LA PLACA ARDUINO UNO

Ya se ha comentado anteriormente que existen varios tipos de placas Arduino, cada una con características específicas que hay que conocer para poder elegir el modelo que más nos convenga según el caso. No obstante, existe un modelo "estándar" de placa, que es el más utilizado con diferencia y que es el que utilizaremos también nosotros en este libro en todos los proyectos: la placa Arduino UNO. Desde que apareció en 2010 ha sufrido tres revisiones, por lo que el modelo actual se suele llamar UNO Rev3 o simplemente UNO R3.



El encapsulado del microcontrolador

La imagen anterior muestra la placa Arduino UNO en su variante convencional. La imagen siguiente muestra la variante llamada Arduino UNO SMD. La única diferencia entre ambas placas es el encapsulado físico del microcontrolador incorporado: ambas tienen el mismo modelo, pero la placa convencional lo lleva montado en formato DIP ("Dual In-line Package") y la placa SMD lo lleva en formato SMD ("Surface Mount Device"). Tal como se puede apreciar en ambas figuras, el formato DIP (visualmente, un gran rectángulo en el centro-inferior-derecha de la placa) es mucho más grande que el formato SMD (visualmente, un pequeño cuadrado ubicado en diagonal en el centro-inferior-derecha de la placa).



Una diferencia importante entre el formato SMD y el DIP es que el primero está soldado a la superficie de la placa (mediante una tecnología llamada precisamente "de montaje superficial" –en inglés, SMT, de "surface mount technology" –), mientras que el segundo está conectado a la placa mediante una serie de patillas metálicas (las cuales son, de hecho, las patillas de E/S del microcontrolador) que se pueden separar fácilmente y que permiten la substitución del microcontrolador por otro si fuera necesario. En la práctica, esto no nos debería importar demasiado a no ser que deseemos separar y reutilizar el microcontrolador de nuestra placa en otras placas o montajes; en ese caso, deberíamos optar por el formato DIP.

Aunque en la placa Arduino solamente dispongamos para nuestro microcontrolador de estas dos alternativas de encapsulado (DIP o SMD), el mundo de los chips en general no es tan sencillo, ya que existen muchas variantes de los dos encapsulados anteriores, además de otros tipos de encapsulados diferentes. Esto es debido a las diferentes necesidades que existen respecto la disponibilidad de espacio físico y la disposición de los conectores en las PCBs que utilicemos. Si se desea consultar qué encapsulados son los más importantes en el mundo electrónico, se puede descargar de http://goo.gl/OU47S un breve documento que los resume. Si aún se desea obtener una información más exhaustiva, recomiendo consultar la página http://www.siliconfareast.com/ic-package-types.htm.

De todas formas, en los proyectos de este libro no nos preocuparemos demasiado por los encapsulados de los chips utilizados, ya que para trabajar con ellos haremos uso de placas "breakout". Una placa "breakout" es una placa PCB que lleva soldado un chip (o más) junto con los conectores y la circuitería necesaria para permitir enchufar a ella dispositivos externos de una forma fácil y rápida. Por tanto, cuando necesitemos en nuestros proyectos algún chip en particular, recurriremos a alguna placa breakout que lo incorpore y entonces simplemente deberemos utilizar los conectores ofrecidos por ella para poner el chip en comunicación con el resto del circuito. Así pues, solo en el caso de necesitar soldar un chip individual a una placa es cuando su tipo de encapsulado lo deberíamos tener en cuenta, pero esto es un tema avanzado que no abordaremos.

Aclaremos que otros componentes simples que no son circuitos integrados (tales como resistencias, condensadores, diodos, fusibles, etc.) también pueden estar encapsulados en formato SMD para optimizar al máximo el espacio físico ocupado. En estos casos, la variedad de formas y tamaños, aunque estandarizada, es inmensa, y se sale de los objetivos de este libro profundizar en este tema. Baste decir que

muchos de estos encapsulados se suelen distinguir por un código de cuatro dígitos, los dos primeros de los cuales indican la longitud del componente y los dos últimos su anchura, en centésimas de pulgadas. Por ejemplo, el encapsulado (por otra parte muy común) “0603” indicaría que el componente en cuestión tiene un tamaño de 0,06” x 0,03”. Otros encapsulados comunes son el “0805” y el “0402”.

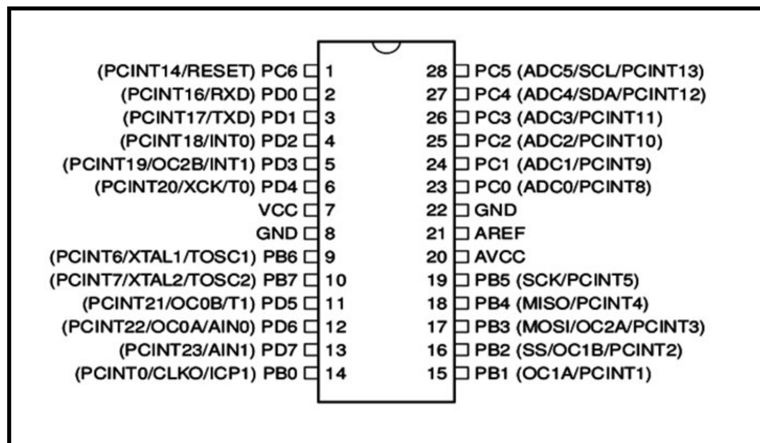
El modelo del microcontrolador

El microcontrolador que lleva la placa Arduino UNO es el modelo ATmega328P de la marca Atmel. La “P” del final significa que este chip incorpora la tecnología “Picopower” (propietaria de Atmel), la cual permite un consumo eléctrico sensiblemente menor comparándolo con el modelo equivalente sin “Picopower”, el Atmega328 (sin la “P”). De todas formas, aunque el ATmega328P pueda trabajar a un voltaje menor y consumir menos corriente que el Atmega328 (especialmente en los modos de hibernación), ambos modelos son funcionalmente idénticos.

Al igual que ocurre con el resto de microcontroladores usados en otras placas Arduino, el ATmega328P tiene una arquitectura de tipo AVR, arquitectura desarrollada por Atmel y en cierta medida “competencia” de otras arquitecturas como por ejemplo la PIC del fabricante Microchip. Más concretamente, el ATmega328P pertenece a la subfamilia de microcontroladores “megaAVR”. Otras subfamilias de la arquitectura AVR son la “tinyAVR” (cuyos microcontroladores son más limitados y se identifican con el nombre de ATtiny) y la “XMEGA” (cuyos microcontroladores son más capaces y se identifican con el nombre de ATxmega), pero no las estudiaremos ya que las placas Arduino no incorporan microcontroladores de esas familias.

De todas formas, si se desea saber más sobre la arquitectura AVR y los modelos y características que ofrecen los microcontroladores construidos de esta forma, no hay nada mejor como consultar la web de la propia empresa fabricante: http://www.atmel.com/products/microcontrollers/avr/default.aspx). Y más en concreto, si se desea consultar la especificación técnica del ATmega328P (aunque para los proyectos de este libro no será necesario) se puede descargar aquí: http://www.atmel.com/dyn/resources/prod_documents/doc8161.pdf .

Lo que sí nos puede venir bien es conocer la disposición concreta de las patillas (llamadas también "pines") de entrada/salida del microcontrolador, ya que, aunque hemos dicho anteriormente que en general todos los pines de E/S sirven para comunicar el microcontrolador con el mundo exterior, es cierto que cada pin suele tener una determinada función específica. Como cada modelo de microcontrolador tiene un número y ubicación de pines diferente, en nuestro caso concreto deberemos tener a mano la disposición de pines del ATmega328P. La figura siguiente muestra esta disposición en el encapsulado de tipo DIP, y ha sido obtenida de la especificación técnica mencionada en el párrafo anterior. Nota: el circulito que aparece en la parte superior de la figura indica el lugar donde existe una muesca en el encapsulado real, de manera que así sea fácil distinguir la orientación de los pines.



Observando la imagen se puede saber qué pin es el que recibe la alimentación eléctrica (señalado como "VCC"), qué dos pines están conectados a tierra (los señalados como "GND"), qué pines son los de E/S (señalados como PBx, PCx o PDx) y la existencia de otros pines más específicos como el AVCC (donde se recibe la alimentación suplementaria para el convertidor analógico-digital interno del chip) o el AREF (donde se recibe la referencia analógica para dicho convertidor –esto lo estudiaremos más adelante–). También se puede observar que junto el nombre de los pines de E/S se indica entre paréntesis las funciones especializadas que cada uno de ellos tiene en particular (además de su función genérica de entrada/salida). Algunas de estas funciones específicas las iremos estudiando a lo largo del libro, como por ejemplo la función de "reset" del microcontrolador, o la comunicación con el exterior usando el protocolo serie o el SPI o el I^2C, o el uso de interrupciones, o el de las salidas PWM, etc.

Las memorias del microcontrolador

Otra cosa que hay que saber de los microcontroladores son los tipos y cantidades de memoria que alojan en su interior. En el caso del ATmega328P tenemos:

Memoria Flash: memoria persistente donde se almacena permanentemente el programa que ejecuta el microcontrolador (hasta una nueva reescritura si se da el caso). En el caso del ATmega328P tiene una capacidad de 32KB.

Breve nota sobre las unidades de medida de la información:

Es buen momento para repasar cómo se mide la cantidad de datos almacenados en una memoria (o transferidos por algún canal). El sistema de medición de la información es el siguiente:

1 bit es la unidad mínima, y puede valer 0 o 1.
1 byte es un grupo de 8 bits.
1 kilobyte (a veces escrito como KB) es un grupo de 1024 bytes (es decir, 8192 bits).
1 megabyte (MB) es un grupo de 1024 KB (es decir, 1048576 bytes).
1 gigabyte (GB) es un grupo de 1024 MB.

Observar que los múltiplos "kilo", "mega" y "giga" hacen referencia a agrupaciones de 1024 (2^{10}) elementos, en vez de lo que suele ocurrir con el resto de unidades de medida del mundo físico, donde son agrupaciones de 1000 elementos. Esta discrepancia es debe a la naturaleza intrínsecamente binaria de los componentes electrónicos.

Observar también que no es lo mismo Kbit ("kilobit") que KB ("kilobyte"). En el primer caso estaríamos hablando de 1024 bits, y en el segundo de 1024 bytes (es decir, 8192 bits, ocho veces más).

En los microcontroladores que vienen incluidos en la placa Arduino no podemos usar toda la capacidad de la memoria Flash porque existen 512 bytes (el llamado "bootloader block") ocupados ya por un código preprogramado de fábrica (el llamado "bootloader" o "gestor de arranque"), el cual nos permite usar la placa Arduino de una forma sencilla y cómoda sin tener que conocer las interioridades electrónicas más avanzadas del microcontrolador. Los ATmega328P que podamos adquirir individualmente normalmente no incluyen de fábrica este pequeño programa, por lo que sí que ofrecen los 32 KB íntegros, pero a cambio no podremos

esperar conectarlos a una placa Arduino y que funcionen sin más ya que les faltará tener grabada esa "preconfiguración". De los gestores de arranque, de su uso y su importancia hablaremos en un próximo apartado.

Memoria SRAM: memoria volátil donde se alojan los datos que en ese instante el programa (grabado separadamente en la memoria Flash, recordemos) necesita crear o manipular para su correcto funcionamiento. Estos datos suelen tener un contenido variable a lo largo del tiempo de ejecución del programa y cada uno es de un tipo concreto (es decir, un dato puede contener un valor numérico entero, otro un número decimal, otro un valor de tipo carácter… también pueden ser cadenas de texto fijas u otros tipos de datos más especiales). Independientemente del tipo de dato, su valor siempre será eliminado cuando se deje de alimentar eléctricamente al microcontrolador. En el caso del ATmega328P esta memoria tiene una capacidad de 2KB.

Si necesitáramos ampliar la cantidad de memoria SRAM disponible, siempre podríamos adquirir memorias SRAM independientes y conectarlas al microcontrolador utilizando algún protocolo de comunicación conocido por este (como SPI o I^2C, de los cuales hablaremos enseguida); no obstante, esto no será necesario en los proyectos de este libro.

Memoria EEPROM: memoria persistente donde se almacenan datos que se desea que permanezcan grabados una vez apagado el microcontrolador para poderlos usar posteriormente en siguientes reinicios. En el caso del ATmega328P esta memoria tiene una capacidad de 1 KB, por lo que se puede entender como una tabla de 1024 posiciones de un byte cada una.

Si necesitáramos ampliar la cantidad de memoria EEPROM disponible, siempre podemos adquirir memorias EEPROM independientes y conectarlas al microcontrolador utilizando algún protocolo de comunicación conocido por este (como SPI o I^2C, de los cuales hablaremos enseguida). O bien, alternativamente, adquirir tarjetas de memoria de tipo SD ("Secure Digital") y comunicarlas mediante un circuito específico al microcontrolador. Las memorias SD son en realidad simples memorias Flash, encapsuladas de una forma concreta; son ampliamente utilizadas en cámaras digitales de foto/vídeo y en teléfonos móviles de última generación, ya que ofrecen muchísima capacidad (varios gigabytes) a un precio barato. La razón por la cual este tipo de tarjetas son capaces de ser reconocidas por el ATmega328P es porque pueden funcionar utilizando el protocolo de comunicación SPI.

Podemos deducir de los párrafos anteriores que la arquitectura a la que pertenece el chip ATmega328P (y en general, toda la familia de microcontroladores AVR) es de tipo Harvard. En este tipo de arquitectura, la memoria que aloja los datos (en nuestro caso, la SRAM o la EEPROM) está separada de la memoria que aloja las instrucciones (en nuestro caso, la Flash), por lo que ambas memorias se comunican con la CPU de forma totalmente independiente y en paralelo, consiguiendo así una mayor velocidad y optimización. Otro tipo de arquitectura (que es la que vemos en los PCs) es la arquitectura Von Neumann, en la cual la CPU está conectada a una memoria RAM única que contiene tanto las instrucciones del programa como los datos, por lo que la velocidad de operación está limitada (entre otras cosas) por el efecto cuello de botella que significa un único canal de comunicación para datos e instrucciones.

Los registros del microcontrolador

Los registros son espacios de memoria existentes dentro de la propia CPU de un microcontrolador. Son muy importantes porque tienen varias funciones imprescindibles: sirven para albergar los datos (cargados previamente desde la memoria SRAM o EEPROM) necesarios para la ejecución de las instrucciones previstas próximamente (y así tenerlos perfectamente disponibles en el momento adecuado); sirven también para almacenar temporalmente los resultados de las instrucciones recientemente ejecutadas (por si se necesitan en algún instante posterior) y sirven además para alojar las propias instrucciones que en ese mismo momento estén ejecutándose.

Su tamaño es muy reducido: tan solo tienen capacidad para almacenar unos pocos bits cada uno. Pero este factor es una de las características más importantes de cualquier microcontrolador, ya que cuanto mayor sea el número de bits que "quepan" en sus registros, mayores serán sus prestaciones, en cuanto a poder de cómputo y velocidad de ejecución. En efecto, es fácil ver (simplificando mucho) que un microcontrolador con registros el doble de grandes que otro podrá procesar el doble de cantidad de datos y por tanto, trabajar el "doble de rápido" aun funcionando los dos al mismo ritmo. De hecho, es tan importante esta característica que cuando escuchamos que un microcontrolador es de "8 bits" o de "32 bits", nos estamos refiriendo precisamente a este dato: al tamaño de sus registros.

Dependiendo de la utilidad que vayamos a darle al microcontrolador, será necesario utilizar uno con un tamaño de registros suficiente. Por ejemplo, el control de un electrodoméstico sencillo como una batidora no requiere más que un microcontrolador de 4 u 8 bits. En cambio, el sistema de control electrónico del

motor de un coche o el de un sistema de frenos ABS se basan normalmente en un microcontrolador de 16 o 32 bits.

El chip ATmega328P es de 8 bits. De hecho, todos los microcontroladores que incorporan las diferentes placas Arduino son de 8 bits excepto el incorporado en la placa Arduino Due, que es de 32 bits.

Los protocolos de comunicación I²C/TWI y SPI

Cuando se desea transmitir un conjunto de datos desde un componente electrónico a otro, se puede hacer de múltiples formas. Una de ellas es estableciendo una comunicación “serie”; en este tipo de comunicación la información es transmitida bit a bit (uno tras otro) por un único canal, enviando por tanto un solo bit en cada momento. Otra manera de transferir datos es mediante la llamada comunicación “paralela”, en la cual se envían varios bits simultáneamente, cada uno por un canal separado y sincronizado con el resto.

El microcontrolador, a través de algunos de sus pines de E/S, utiliza el sistema de comunicación serie para transmitir y recibir órdenes y datos hacia/desde otros componentes electrónicos. Esto es debido sobre todo a que en una comunicación serie solo se necesita en teoría un único canal (un único “cable”), mientras que en una comunicación en paralelo se necesitan varios cables, con el correspondiente incremento de complejidad, tamaño y coste del circuito resultante.

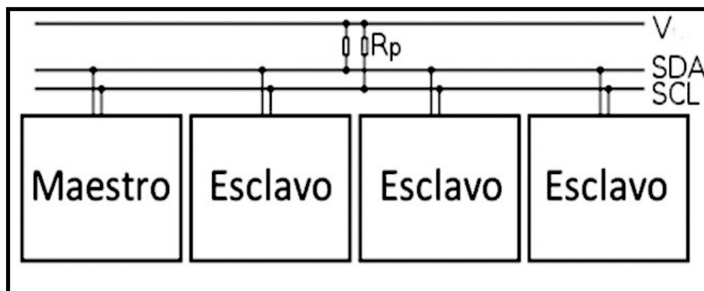
No obstante, no podemos hablar de un solo tipo de comunicación serie. Existen muchos protocolos y estándares diferentes basados todos ellos en la transferencia de información en serie, pero implementando de una forma diferente cada uno los detalles específicos (como el modo de sincronización entre emisor y receptor, la velocidad de transmisión, el tamaño de los paquetes de datos, los mensajes de conexión y desconexión y de dar paso al otro en el intercambio de información, los voltajes utilizados, etc.). De entre el gran número de protocolos de comunicación serie reconocidos por la inmensa variedad de dispositivos electrónicos del mercado, los que nos interesarán conocer son los que el ATmega328P es capaz de comprender y por tanto, los que podrá utilizar para contactar con esa variedad de periféricos. En este sentido, los estándares más importantes son:

I²C (Inter-Integrated Circuit, también conocido con el nombre de **TWI** –de “TWo-wIre”, literalmente “dos cables” en inglés–): es un sistema muy utilizado en la industria principalmente para comunicar circuitos integrados entre sí. Su principal característica es que utiliza dos líneas para transmitir la

información: una (llamada línea “SDA”) sirve para transferir los datos (los 0s y los 1s) y otra (llamada línea “SCL”) sirve para enviar la señal de reloj. En realidad también se necesitarían dos líneas más: la de alimentación y la de tierra común, pero estas ya se presuponen existentes en el circuito.

Por “señal de reloj” se entiende una señal binaria de una frecuencia periódica muy precisa que sirve para coordinar y sincronizar los elementos integrantes de una comunicación (es decir, los emisores y receptores) de forma que todos sepan cuándo empieza, cuánto dura y cuándo acaba la transferencia de información. En hojas técnicas y diagramas a la señal de reloj en general se le suele describir como CLK (del inglés “clock”).

Cada dispositivo conectado al bus I²C tiene una dirección única que lo identifica respecto el resto de dispositivos, y puede estar configurado como “maestro” o como “esclavo”. Un dispositivo maestro es el que inicia la transmisión de datos y además genera la señal de reloj, pero no es necesario que el maestro sea siempre el mismo dispositivo: esta característica se la pueden ir intercambiando ordenadamente los dispositivos que tengan esa capacidad.



Tal como se muestra en el diagrama anterior, para funcionar correctamente tanto la línea “SDA” como la “SCL” necesitan estar conectadas mediante una resistencia “pull-up” a la fuente de alimentación común, la cual puede proveer un voltaje generalmente de 5 V o 3,3 V (aunque sistemas con otros voltajes pueden ser posibles).

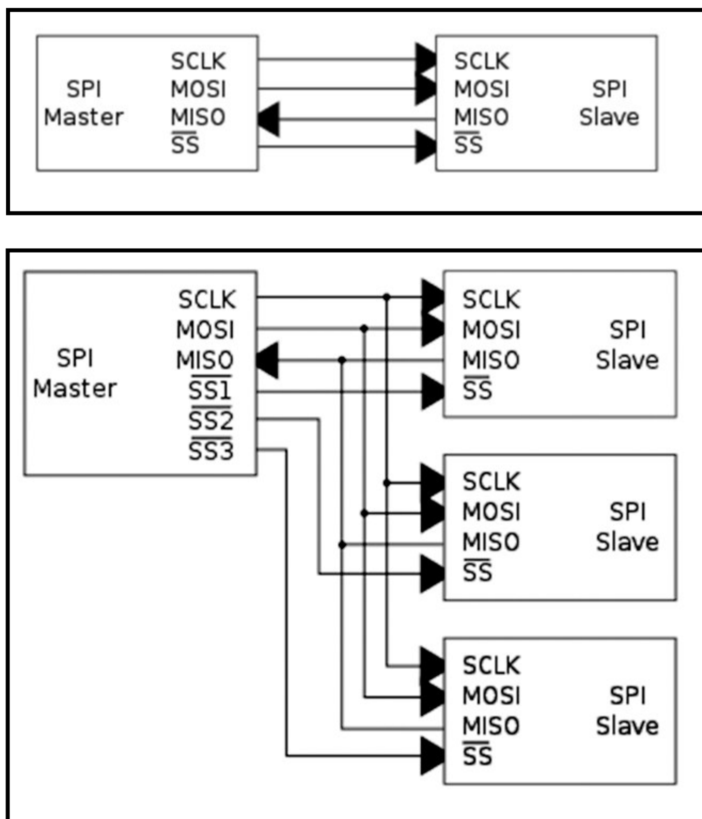
La velocidad de transferencia de datos es de 100 Kbits por segundo en el modo estándar (aunque también se permiten velocidades de hasta 3,4 Mbit/s). No obstante, al haber una única línea de datos, la transmisión de información es “half duplex” (es decir, la comunicación solo se puede establecer en un sentido al mismo

tiempo) por lo que en el momento que un dispositivo empiece a recibir un mensaje, tendrá que esperar a que el emisor deje de transmitir para poder responderle.

SPI (Serial Peripheral Interface): al igual que el sistema I²C, el sistema de comunicación SPI es un estándar que permite controlar (a cortas distancias) casi cualquier dispositivo electrónico digital que acepte un flujo de bits serie sincronizado (es decir, regulado por un reloj). Igualmente, un dispositivo conectado al bus SPI puede ser "maestro" –en inglés, "master" – o "esclavo" –en inglés, "slave"–, donde el primero es el que inicia la transmisión de datos y además genera la señal de reloj (aunque, como con I²C , con SPI tampoco es necesario que el maestro sea siempre el mismo dispositivo) y el segundo se limita a responder.

La mayor diferencia entre el protocolo SPI y el I²C es que el primero requiere de cuatro líneas ("cables") en vez de dos. Una línea (llamada normalmente "SCK") envía a todos los dispositivos la señal de reloj generada por el maestro actual; otra (llamada normalmente "SS") es la utilizada por ese maestro para elegir en cada momento con qué dispositivo esclavo se quiere comunicar de entre los varios que puedan estar conectados (ya que solo puede transferir datos con un solo esclavo a la vez); otra (llamada normalmente "MOSI") es la línea utilizada para enviar los datos –0s y 1s– desde el maestro hacia el esclavo elegido; y la otra (llamada normalmente "MISO") es la utilizada para enviar los datos en sentido contrario: la respuesta de ese esclavo al maestro. Es fácil ver que, al haber dos líneas para los datos la transmisión de información es "full duplex" (es decir, que la información puede ser transportada en ambos sentidos a la vez).

En las siguientes figuras se muestra el esquema de líneas de comunicación existentes entre un maestro y un esclavo y entre un maestro y tres esclavos respectivamente. Se puede observar que, para el caso de la existencia de varios esclavos es necesario utilizar una línea "SS" diferente por cada uno de ellos, ya que esta línea es la que sirve para activar el esclavo concreto que en cada momento el maestro desee utilizar (esto no pasa con las líneas de reloj, "MOSI" y "MISO", que son compartidas por todos los dispositivos).Técnicamente hablando, el esclavo que reciba por su línea SS un valor de voltaje BAJO será el que esté seleccionado en ese momento por el maestro, y los que reciban el valor ALTO no lo estarán (de ahí el subrayado superior que aparece en la figura).



Como se puede ver, el protocolo SPI respecto el I²C tiene la desventaja de exigir al microcontrolador dedicar muchos más pines de E/S a la comunicación externa. En cambio, como ventaja podemos destacar que es más rápido y consume menos energía que I²C.

Tal como se puede observar en la figura que muestra la disposición de pines del microcontrolador ATmega328P (página 75), los pines correspondientes a las líneas I²C SDA y SCL son los números 27 y 28, respectivamente, y los pines correspondientes a las líneas SPI SS, MOSI, MISO y SCK son los números 16, 17, 18 y 19, respectivamente. Si se necesitaran más líneas SS (porque haya más de un dispositivo esclavo conectado en nuestro circuito), se podría utilizar cualquier otro pin de E/S siempre que respete el convenio de poner el valor de su voltaje de salida a BAJO cuando se desee trabajar con el dispositivo esclavo asociado y poner a ALTO el resto de pines SS.

El gestor de arranque del microcontrolador

Ya hemos comentado anteriormente que dentro de la memoria Flash del microcontrolador incluido en las placas Arduino viene pregrabado de fábrica un pequeño programa llamado "bootloader" o "gestor de arranque", que resulta imprescindible para un cómodo y fácil manejo de la placa en cuestión. Este software (también llamado "firmware", porque es un tipo de software que raramente se modifica) ocupa, en la placa Arduino UNO, 512 bytes de espacio en un apartado especial de la memoria Flash, el llamado "bootloader block", pero en otros modelos de placas Arduino puede ocupar más (por ejemplo, en el modelo Leonardo ocupa 4 Kilobytes).

La función de este firmware es gestionar de forma automática el proceso de grabación en la memoria Flash del programa que queremos que el microcontrolador ejecute. Lógicamente, el bootloader realizará esta grabación más allá del "bootloader block" para no sobrescribirse a sí mismo.

Más concretamente, el bootloader se encarga de recibir nuestro programa de parte del entorno de desarrollo Arduino (normalmente mediante una transmisión realizada a través de conexión USB desde el computador donde se está ejecutando dicho entorno hasta la placa) para proceder seguidamente a su correcto almacenamiento en la memoria Flash, todo ello de forma automática y sin que nos tengamos que preocupar de las interioridades electrónicas del proceso. Una vez realizado el proceso de grabación, el bootloader termina su ejecución y el microcontrolador se dispone a procesar de inmediato y de forma permanente (mientras esté encendido) las instrucciones recientemente grabadas.

En la placa Arduino UNO, el bootloader siempre se ejecuta durante el primer segundo de cada reinicio. Durante esos instantes, el gestor de arranque se espera a recibir una serie de instrucciones concretas de parte del entorno de desarrollo para interpretarlas y realizar la correspondiente carga de un posible programa. Si esas instrucciones no llegan pasado ese tiempo, el bootloader termina su ejecución e igualmente se empieza a procesar lo que haya en ese momento en la memoria Flash.

Estas instrucciones internas de programación de memorias Flash son ligeramente diferentes según el tipo de bootloader que tenga la placa, pero casi todas son variantes del conjunto de instrucciones ofrecido oficialmente por Atmel para la programación de sus microcontroladores, el llamado protocolo STK500 (http://www.atmel.com/tools/STK500.aspx). Un ejemplo es el bootloader que tiene pregrabado el ATmega328P del Arduino UNO, basado en un firmware libre llamado

Optiboot (http://code.google.com/p/optiboot), el cual logra una velocidad de grabación de 115 kilobits de programa a cargar por segundo gracias al uso de instrucciones propias derivadas del "estándar" STK500. Otro ejemplo de bootloader derivado del protocolo STK500 es el bootloader "wiring", grabado de fábrica en el microcontrolador de la placa Arduino Mega. El bootloader que viene en la placa Leonardo (llamado "Caterina") es diferente, ya que entiende otro conjunto de instrucciones independiente llamado AVR109 (también oficial de Atmel). Toda esta información se puede obtener consultando el contenido del fichero llamado "boards.txt", descargado junto con el entorno de desarrollo de Arduino.

Si adquirimos un microcontrolador ATmega328P por separado, hay que tener en cuenta que no dispondrá del bootloader, por lo que deberemos incorporarle uno nosotros "a mano" para hacer uso de él a partir de entonces, o bien no utilizar nunca ningún bootloader y cargar entonces siempre nuestros programas a la memoria Flash directamente. En ambos casos, el procedimiento requiere el uso de un aparato específico (en concreto, lo que se llama un "programador ISP" –In System Programmer–) que debemos adquirir aparte. Este aparato se ha de conectar por un lado a nuestro computador y por otro a la placa Arduino, y suple la ausencia de bootloader haciendo de intermediario entre nuestro entorno de desarrollo y la memoria Flash del microcontrolador. Por lo tanto, podemos resumir diciendo que el gestor de arranque es el elemento que permite programar nuestro Arduino directamente con un simple cable USB y nada más.

Por conveniencia, dentro del paquete instalador del entorno de desarrollo de Arduino (descargable de su web oficial, para más detalles ver el capítulo siguiente), se distribuyen además copias exactas bit a bit de los bootloaders oficiales que vienen grabados en los diferentes microcontroladores Arduino. Estas copias exactas son ficheros con extensión ".hex" que tienen un formato interno llamado "Intel Hex Format". Para el uso normal de nuestra placa no necesitamos para nada estos ficheros ".hex", pero si dispusiéramos de un programador ISP y en algún momento tuviéramos que "reponer" un bootloader dañado (o bien grabar un bootloader a algún microcontrolador que no tuviera ninguno), Arduino nos ofrece estos ficheros para cargarlos en la memoria Flash de nuestro microcontrolador siempre que queramos.

El formato Intel Hex Format es el utilizado por todos los chips AVR para almacenar contenido en sus memorias Flash, por lo que hay que aclarar que no solamente los bootloaders son alojados internamente de esta forma en la memoria Flash, sino que todos nuestros propios programas que escribamos en el entorno de desarrollo también serán alojados allí en formato ".hex" (aunque de estos detalles no nos debemos preocupar por ahora).

Evidentemente, los bootloaders Arduino también son software libre, por lo que al igual que ocurre con el entorno de programación Arduino, siempre tendremos disponible su código fuente (escrito en lenguaje C) para poder conocer cómo funciona internamente e incluso para poderlo modificar, si así se estima oportuno.

¿QUÉ OTRAS CARACTERÍSTICAS TIENE LA PLACA ARDUINO UNO?

La placa Arduino UNO, aparte del microcontrolador que incorpora, tiene otras características interesantes a repasar:

La alimentación

El voltaje de funcionamiento de la placa Arduino (incluyendo el microcontrolador y el resto de componentes) es de 5 V. Podemos obtener esta alimentación eléctrica de varias maneras:

Conectando la placa Arduino a una fuente externa, tal como un adaptador AC/DC o una pila. Para el primer caso, la placa dispone de un zócalo donde se puede enchufar una clavija de 2,1 milímetros de tipo “jack”. Para el segundo, los cables salientes de los bornes de la pila se pueden conectar a los pines-hembra marcados como “Vin” y “Gnd” (positivo y negativo respectivamente) en la zona de la placa marcada con la etiqueta “POWER”. En ambos casos, la placa está preparada en teoría para recibir una alimentación de 6 a 20 voltios, aunque, realmente, el rango recomendado de voltaje de entrada (teniendo en cuenta el deseo de obtener una cierta estabilidad y seguridad eléctricas en nuestros circuitos) es menor: de 7 a 12 voltios. En cualquier caso, este voltaje de entrada ofrecido por la fuente externa siempre es rebajado a los 5 V de trabajo mediante un circuito regulador de tensión que ya viene incorporado dentro de la placa.

Conectando la placa Arduino a nuestro computador mediante un cable USB. Para ello, la placa dispone de un conector USB hembra de tipo B. La alimentación recibida de esta manera está regulada permanentemente a los 5 V de trabajo y ofrece un máximo de hasta 500 mA de corriente (por lo tanto, la potencia consumida por la placa es en ese caso de unos 2,5 W). Si en algún momento por el conector USB pasa más intensidad de la deseable, la placa Arduino está protegida mediante un polifusible reseteable que automáticamente rompe la conexión hasta que las condiciones eléctricas vuelven a la normalidad. Una consecuencia de esta protección contra posibles picos de corriente es que la intensidad de corriente recibida a través

de USB puede no ser suficiente para proyectos que contengan componentes tales como motores, solenoides o matrices de LEDs, los cuales consumen mucha potencia.

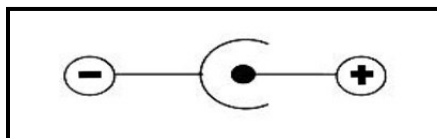
Sea cual sea la manera elegida para alimentar la placa, esta es lo suficientemente "inteligente" para seleccionar automáticamente en cada momento la fuente eléctrica disponible y utilizar una u otra sin que tengamos que hacer nada especial al respecto.

Si utilizamos una pila como alimentación externa, una ideal sería la de 9 V (está dentro del rango recomendado de 7 a 12 voltios), y si se utiliza un adaptador AC/DC, se recomienda el uso de uno con las siguientes características:

El voltaje de salida ofrecido ha de ser de 9 a 12 V DC. En realidad, el circuito regulador que lleva incorporado la placa Arduino es capaz de manejar voltajes de salida (de entrada para la placa) de hasta 20 V, así que en teoría se podrían utilizar adaptadores AC/DC que generen una salida de 20 V DC. No obstante, esta no es una buena idea porque se pierde la mayoría del voltaje en forma de calor (lo cual es terriblemente ineficiente) y además puede provocar el sobrecalentamiento del regulador, y como consecuencia dañar la placa.

La intensidad de corriente ofrecida ha de ser de 250 mA (o más). Si conectamos a nuestra placa Arduino muchos componentes o unos pocos pero consumidores de mucha energía (como por ejemplo una matriz de LEDs, una tarjeta SD o un motor) el adaptador debería suministrar al menos 500 mA o incluso 1 A. De esta manera nos aseguraremos de que tenemos suficiente corriente para que cada componente pueda funcionar de forma fiable.

El adaptador ha de ser de polaridad "con el positivo en el centro". Esto quiere decir que la parte externa del cilindro metálico que forma la clavija de 5,5/2,1 mm del adaptador ha de ser el borne negativo y el hueco interior del cilindro ha de ser el borne positivo. Lo más sencillo para estar seguros de que nuestro adaptador es el adecuado en este sentido es observar si tiene imprimido en algún sitio el siguiente símbolo:



Por otro lado, dentro de la zona etiquetada como “POWER” en la placa Arduino existe una serie de pines-hembra relacionados con la alimentación eléctrica, como son:

”GND”: pines-hembra conectados a tierra. Es muy importante que todos los componentes de nuestros circuitos compartan una tierra común como referencia. Estos pines-hembra se ofrecen para realizar esta función.

”Vin”: este pin-hembra se puede utilizar para dos cosas diferentes: si la placa está conectada mediante la clavija de 2,1mm a alguna fuente externa que aporte un voltaje dentro de los márgenes de seguridad, podemos conectar a este pin-hembra cualquier componente electrónico para alimentarlo directamente con el nivel de voltaje que esté aportando la fuente en ese momento (¡sin regular por la placa!) . Si la placa está alimentada mediante USB, entonces ese pin-hembra aportará 5 V regulados. En cualquier caso, la intensidad de corriente máxima aportada es de 40 mA (esto hay que tenerlo en cuenta cuando conectemos dispositivos que consuman mucha corriente, como por ejemplo motores). También podemos usar el pin-hembra “Vin” para otra cosa: para alimentar la propia placa directamente desde alguna fuente de alimentación externa sin utilizar ni la clavija ni el cable USB. Esto se hace conectándole el borne positivo de la fuente (por ejemplo, una pila de 9 V) y conectando el borne negativo al pin de tierra. Si se usa este montaje, el regulador de tensión que incorpora la placa reducirá el voltaje recibido de la pila al voltaje de trabajo de la placa (los 5 V).

”5 V”: este pin-hembra se puede utilizar para dos cosas diferentes: tanto si la placa está alimentada mediante el cable USB como si está alimentada por una fuente externa que aporte un voltaje dentro de los márgenes de seguridad, podemos conectar a este pin-hembra cualquier componente para que pueda recibir 5 V regulados. En cualquier caso, la intensidad de corriente máxima generada será de 40 mA. Pero también podemos usar este pin-hembra para otra cosa: para alimentar la propia placa desde una fuente de alimentación externa previamente regulada a 5 V sin utilizar el cable USB ni la clavija de 2,1mm.

”3,3 V”: este pin-hembra ofrece un voltaje de 3,3 voltios. Este voltaje se obtiene a partir del recibido indistintamente a través del cable USB o de la clavija de 2,1 mm, y está regulado (con un margen de error del 1%) por un circuito específico incorporado en la placa: el LP2985. En este caso particular, la corriente máxima generada es de 50 mA. Al igual que con los pines anteriores, podremos usar este pin para alimentar componentes de nuestros

circuitos que requieran dicho voltaje (los más delicados), pero en cambio, no podremos conectar ninguna fuente externa aquí porque el voltaje es demasiado limitado para poder alimentar a la placa.

El chip ATmega16U2

La conexión USB de la placa Arduino, además de servir como alimentación eléctrica, sobre todo es un medio para poder transmitir datos entre nuestro computador y la placa, y viceversa. Este tráfico de información que se realiza entre ambos aparatos se logra gracias al uso del protocolo USB, un protocolo de tipo serie que tanto nuestro computador como la placa Arduino son capaces de entender y manejar. No obstante, el protocolo USB internamente es demasiado complejo para que el microcontrolador ATmega328P pueda comprenderlo por sí mismo sin ayuda, ya que él tan solo puede comunicarse con el exterior mediante protocolos mucho más sencillos técnicamente como son el I^2C o el SPI y pocos más. Por tanto, es necesario que la placa disponga de un elemento "traductor" que facilite al ATmega328P (concretamente, al receptor/transmisor serie de tipo TTL-UART que lleva incorporado) la manipulación de la información transferida por USB sin que este tenga que conocer los entresijos de dicho protocolo.

Breve nota sobre la tecnología TTL:

Por "TTL" ("Transistor-to-Transistor Logic") se entiende un tipo genérico de circuito electrónico donde los elementos de entrada y salida de éste son transistores bipolares. Y concretamente el chip UART ("Universal Asynchronous Receiver-Transmitter") que viene incorporado dentro del microcontrolador ATmega328P (y que es el encargado de establecer la comunicación de tipo serie entre este y el exterior) está construido con la tecnología TTL.

Que el receptor/transmisor UART sea de tipo TTL implica que los valores HIGH (bits "1") enviados o transmitidos los representará con un pulso de 5V (o 3,3V, según el voltaje de trabajo del circuito) y los valores LOW (bits "0") se representará con un pulso de 0 V. Esto es importante porque otros tipos de receptores/transmisores también serie (como por ejemplo los de tipo RS-232, que no estudiaremos) utilizan otros niveles de voltaje para representar los valores HIGH y LOW y por tanto no son compatibles.

La placa Arduino UNO R3 dispone de un chip que realiza esta función de "traductor" del protocolo USB a un protocolo serie más sencillo (y viceversa). Ese chip es el ATmega16U2. El ATmega16U2 es todo un microcontrolador en sí mismo (con su propia CPU, con su propia memoria –tiene por ejemplo 16 Kilobytes de

memoria Flash para su uso interno, de ahí su nombre–, etc.) y por tanto podría realizar muchas más tareas que no solo la “traducción” del protocolo USB. De hecho técnicamente es posible desprogramarlo para que haga otras cosas y convertir así la placa Arduino en virtualmente cualquier tipo de dispositivo USB conectado a nuestro computador (un teclado, un ratón, un dispositivo MIDI…). No obstante, por defecto el ATmega16U2 que viene incluido en la placa Arduino viene ya con el firmware preprogramado para realizar exclusivamente la función de “intérprete” al ATmega328P y ya está.

Este firmware es software libre, por lo que se puede acceder a su código fuente y también están disponible su correspondiente fichero “.hex”, dentro del conjunto de ficheros descargados, junto con el entorno de desarrollo oficial de Arduino (concretamente, dentro de la carpeta “firmwares”, dentro de “hardware/arduino”).

En modelos de la placa Arduino anteriores al UNO (como el modelo NG, el Diecimila o el Duemilanove) el chip ATmega16U2 no venía: en su lugar aparecía un circuito conversor de USB a serie del fabricante FTDI, concretamente el FT232RL. Una ventaja de haber sustituido el FT232RL por el ATmega16U2 es el precio. Otra ventaja es el tener la posibilidad (si somos usuarios avanzados, tal como hemos comentado ya) de reprogramar el ATmega16U2 para que en vez de funcionar como un simple conversor USB-Serie pueda simular ser cualquier otro tipo de dispositivo USB, que, por ejemplo, con el FT232RL no podemos porque está pensado para hacer tan solo la función para la que fue construido.

El ATmega16U2 viene acompañado en la placa Arduino por un reloj oscilador de cristal, de uso exclusivo para él, que sirve para mantener la sincronización con la comunicación USB. De los relojes hablaremos en un apartado posterior.

Las entradas y salidas digitales

La placa Arduino dispone de 14 pines-hembra de entradas o salidas (según lo que convenga) digitales, numeradas desde la 0 hasta la 13. Es aquí donde conectaremos nuestros sensores para que la placa pueda recibir datos del entorno, y también donde conectaremos los actuadores para que la placa pueda enviarles las órdenes pertinentes, además de poder conectar cualquier otro componente que necesite comunicarse con la placa de alguna manera. A veces a estos pines-hembra digitales de “propósito general” se les llama pines GPIO (de “General Purpose Input/Output”).

Todos estos pines-hembra digitales funcionan a 5 V, pueden proveer o recibir un máximo de 40 mA y disponen de una resistencia “pull-up” interna de entre 20 KΩ y 50 KΩ que inicialmente está desconectada (salvo que nosotros indiquemos lo contrario mediante programación software).

Hay que tener en cuenta, no obstante, que aunque cada pin individual pueda proporcionar hasta 40 mA como máximo, en realidad, internamente la placa agrupa los pines digitales de tal forma que tan solo pueden aportar 100 mA a la vez el conjunto de los pines nº 0,1,2,3 y 4, y 100 mA más el resto de pines (del 5 al 13). Esto quiere decir que como mucho podríamos tener 10 pines ofreciendo 20 mA a la vez.

Las entradas analógicas

La placa Arduino dispone de 6 entradas analógicas (en forma de pines-hembra etiquetados como “A0”, “A1”... hasta “A5”) que pueden recibir voltajes dentro de un rango de valores continuos de entre 0 y 5 V. No obstante, la electrónica de la placa tan solo puede trabajar con valores digitales, por lo que es necesaria una conversión previa del valor analógico recibido a un valor digital lo más aproximado posible. Esta se realiza mediante un circuito conversor analógico/digital incorporado en la propia placa.

El circuito conversor es de 6 canales (uno por cada entrada) y cada canal dispone de 10 bits (los llamados “bits de resolución”) para guardar el valor del voltaje convertido digitalmente.

En general, la cantidad de bits de resolución que tiene un determinado conversor analógico/digital es lo que marca en gran medida el grado de precisión conseguida en la conversión de señal analógica a digital, ya que cuantos más bits de resolución tenga, más fiel será la transformación. Por ejemplo, en el caso concreto del conversor incorporado en la placa Arduino, si contamos el número de combinaciones de 0s y 1s que se pueden obtener con 10 posiciones, vemos que hay un máximo de 2^{10} (1024) valores diferentes posibles. Por tanto, la placa Arduino puede distinguir para el voltaje digital desde el valor 0 hasta el valor 1023. Si el conversor tuviera por ejemplo 20 bits de resolución, la variedad de valores digitales que podría distinguir sería muchísimo más grande (2^{20} = 1048576) y podría afinar la precisión mucho más.

Esto es fácil verlo si dividimos el rango analógico de entrada (5 V - 0V = 5 V) entre el número máximo posible de valores digitales (1024). Obtendremos que cada valor digital corresponde a una “ventana” analógica de aproximadamente 5

V/1024≈5 mV. En otras palabras: todos los valores analógicos dentro de cada rango de 5 mV (desde 0 a 5 V) se “colapsan” sin distinción en un único valor digital (desde 0 a 1023). Así pues, no podremos distinguir valores analógicos distanciados por menos de 5 mV.

En muchos de nuestros proyectos ya nos es suficiente este grado de precisión, pero en otros puede que no. Si el conversor analógico/digital tuviera más bits de resolución, el resultado de la división *rango_analógico_entrada/número_valores_digitales* sería menor, y por tanto la conversión sería más rigurosa. Pero como no se pueden aumentar los bits de resolución del conversor de la placa, si queremos más exactitud se ha de optar por otra solución: en vez de aumentar el denominador de la división anterior, se puede reducir su numerador (es decir, el rango analógico de entrada, o más específicamente, su límite superior –por defecto igual a 5 V–, ya que el inferior es 0). Este límite superior en la documentación oficial se suele nombrar como “voltaje de referencia”.

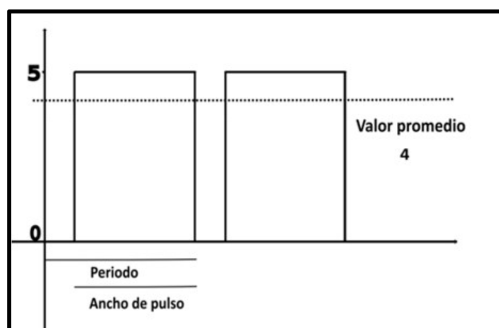
La manera práctica y concreta de reducir el voltaje de referencia del conversor analógico/digital de la placa Arduino no la podemos explicar todavía porque aún nos faltan los conocimientos necesarios para poder llevar a cabo todo el proceso. Será explicado en el apartado correspondiente del capítulo 6.

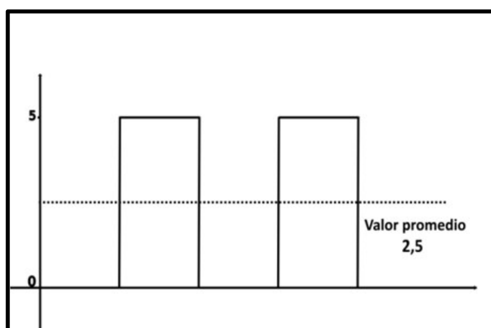
Por último, decir que estos pines-hembra de entrada analógica tienen también toda la funcionalidad de los pines de entrada-salida digitales. Es decir, que si en algún momento necesitamos más pines-hembra digitales más allá de los 14 que la placa Arduino ofrece (del 0 al 13), los 6 pines-hembra analógicos pueden ser usados como unos pines-hembra digitales más (numerándose entonces del 14 al 19) sin ninguna distinción.

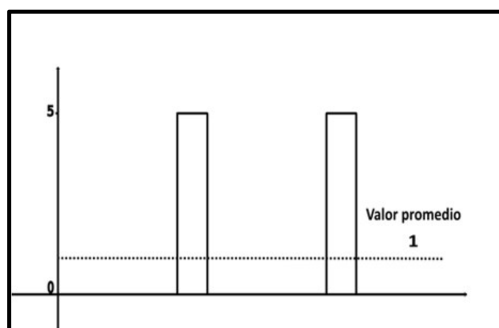
Las salidas analógicas (PWM)

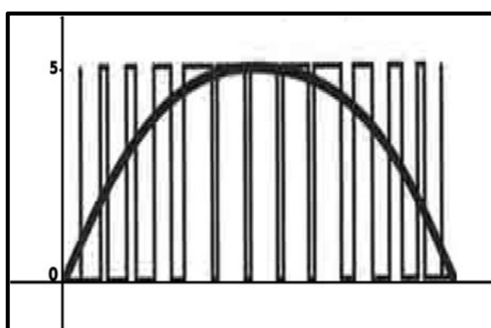
En nuestros proyectos a menudo necesitaremos enviar al entorno señales analógicas, por ejemplo, para variar progresivamente la velocidad de un motor, la frecuencia de un sonido emitido por un zumbador o la intensidad con la que luce un LED. No basta con simples señales digitales: tenemos que generar señales que varíen continuamente. La placa Arduino no dispone de pines-hembra de salida analógica propiamente dichos (porque su sistema electrónico interno no es capaz de manejar este tipo de señales), sino que utiliza algunos pines-hembra de salida digitales concretos para “simular” un comportamiento analógico. Los pines-hembra digitales que son capaces trabajar en este modo no son todos: solo son los marcados con la etiqueta “PWM”. En concreto para el modelo Arduino UNO son los pines número: 3, 5, 6, 9, 10 y 11.

Las siglas PWM vienen de “Pulse Width Modulation” (Modulación de Ancho de Pulso). Lo que hace este tipo de señal es emitir, en lugar de una señal continua, una señal cuadrada formada por pulsos de frecuencia constante (aproximadamente de 490 Hz). La gracia está en que al variar la duración de estos pulsos, estaremos variando proporcionalmente la tensión promedio resultante. Es decir: cuanto más cortos sean los pulsos (y por tanto, más distantes entre sí en el tiempo, ya que su frecuencia es constante), menor será la tensión promedio de salida, y cuanto más largos sean los pulsos (y por tantos, más juntos en el tiempo estén), mayor será dicha tensión. El caso extremo lo tendríamos cuando la duración del pulso coincidiera con el período de la señal, momento en el cual de hecho no habría distancia entre pulso y pulso (sería una señal de un valor constante) y la tensión promedio de salida sería la máxima posible, que son 5 V. La duración del pulso la podemos cambiar en cualquier momento mientras la señal se está emitiendo, por lo que como consecuencia la tensión promedio puede ir variando a lo largo del tiempo de forma continua. Las siguientes figuran ilustran lo acabado de explicar:









Cada pin-hembra PWM tiene una resolución de 8 bits. Esto quiere decir que si contamos el número de combinaciones de 0s y 1s que se pueden obtener con 8 posiciones, obtendremos un máximo de 2^8 (256) valores diferentes posibles. Por

tanto, podemos tener 256 valores diferentes para indicar la duración deseada de los pulsos de la señal cuadrada (o dicho de otra forma: 256 valores promedio diferentes). Si establecemos (mediante programación software) el valor mínimo (0), estaremos emitiendo unos pulsos extremadamente estrechos y generaremos una señal analógica equivalente a 0 V; si establecemos el valor máximo (255), estaremos emitiendo pulsos de máxima duración y generaremos una señal analógica equivalente a 5 V. Cualquier valor entremedio emitirá pulsos de duración intermedia y por tanto, generará una señal analógica de un valor entre 0 V y 5 V.

La diferencia de voltaje analógico existente entre dos valores promedio contiguos (es decir, entre por ejemplo el valor número 123 y el número 124) se puede calcular mediante la división: *rango_voltaje_salida/número_valores_promedio*. En nuestro caso, sería (5 V – 0 V)/256 ≈19,5 mV. Es decir, cada valor promedio está distanciado del anterior y del siguiente por un "saltito" de 19,5 mV.

Es posible cambiar la frecuencia por defecto de la señal cuadrada utilizada en la generación de la señal "analógica", pero no es un procedimiento trivial, y en la mayoría de ocasiones no nos será necesario. En este sentido, tan solo comentaremos que los pines PWM viene controlados por tres temporizadores diferentes que mantienen la frecuencia constante de los pulsos emitidos; concretamente, los pines 3 y 11 son controlados por el "Timer1", los pines 5 y 6 por el "Timer2" y los pines 9 y 10 por el "Timer3".

Otros usos de los pines-hembra de la placa

Existen determinados pines-hembra de entrada/salida digitales, que además de su función "estándar", tienen otras funciones especializadas. Por ejemplo:

Pin 0 (RX) y pin 1 (TX): permiten que el microcontrolador ATmega328P pueda recibir directamente datos en serie (por el pin RX) o transmitirlos (por el pin TX) sin pasar por la conversión USB-Serie que realiza el chip ATmega16U2. Es decir, estos pines posibilitan la comunicación sin intermediarios de dispositivos externos con el receptor/transmisor serie (de tipo TTL-UART) que incorpora el propio ATmega328P. De todas maneras, estos pines están internamente conectados (mediante resistencias de 1 KΩ) al chip ATmega16U2, por lo que los datos disponibles en el USB también lo estarán en estos pines.

Hay que aclarar que en la placa están incrustados un par de LEDs etiquetados como "RX" y "TX", pero que, a pesar de su nombre, no se encienden cuando se

reciben o transmiten datos de los pines 0 y 1, sino solamente cuando se reciben o transmiten datos provenientes de la conexión USB a través del chip ATmega16U2.

Pines 2 y 3: se pueden usar, con la ayuda de programación software, para gestionar interrupciones. No obstante, este tema es relativamente avanzado y no lo abordaremos en este libro.

Pines 10 (SS), 11 (MOSI) , 12 (MISO) y 13 (SCK): se pueden usar para conectar algún dispositivo con el que se quiera llevar a cabo comunicaciones mediante el protocolo SPI. Estudiaremos casos concretos más adelante.

Pin 13: este pin está conectado directamente a un LED incrustado en la placa (identificado con la etiqueta "L") de forma que si el valor del voltaje recibido por este pin es ALTO (HIGH), el LED se encenderá, y si dicho valor es BAJO (LOW), el LED se apagará. Es una manera sencilla, y rápida de detectar señales de entradas externas sin necesidad de disponer de ningún componente extra.

También existen un par de pines-hembra de entrada analógica que tienen una función extra además de la habitual:

Pines A4 (SDA) y A5 (SCL): se pueden usar para conectar algún dispositivo con el que se quiera llevar a cabo comunicaciones mediante el protocolo I^2C/TWI. La placa Arduino ofrece (por una simple cuestión de comodidad y ergonomía) una duplicación de estos dos pines-hembra en los dos últimos pines-hembra tras el pin "AREF", los cuales están sin etiquetar porque no hay más espacio físico.

Finalmente, a lo largo de la placa existen diferentes pines-hembra no comentados todavía que no funcionan ni como salidas ni como entradas porque tienen un uso muy específico y concreto:

Pin AREF: ofrece un voltaje de referencia externo para poder aumentar la precisión de las entradas analógicas. Estudiaremos su uso práctico en el capítulo 6.

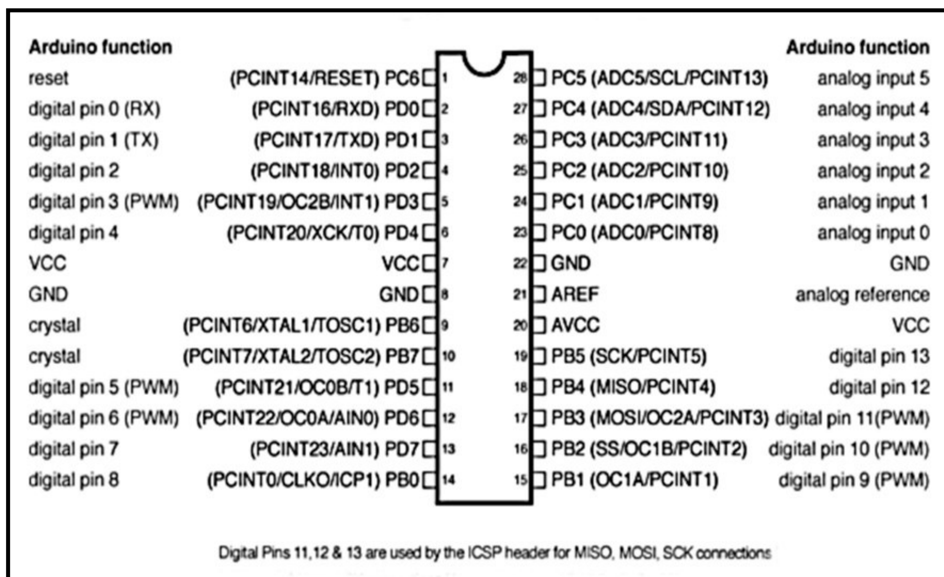
Pin RESET: si el voltaje de este pin se establece a valor BAJO (LOW), el microcontrolador se reiniciará y se pondrá en marcha el bootloader. Para realizar esta misma función, la placa Arduino ya dispone de un botón, pero este pin ofrece la posibilidad de añadir otro botón de reinicio a placas

supletorias (es decir, placas que se conectan encima de la placa Arduino para ampliarla y complementarla), las cuales por su colocación puedan ocultar o bloquear el botón de la placa Arduino.

Pin IOREF: en realidad este pin es una duplicación regulada del pin "Vin". Su función es indicar a las placas supletorias conectadas a nuestra placa Arduino el voltaje al que trabajan los pines de entrada/salida de esta, para que las placas supletorias se adapten automáticamente a ese voltaje de trabajo (que en el caso del modelo UNO ya sabemos que es 5 V).

Pin sin utilizar: justo el pin a continuación del IOREF, el cual está sin etiquetar, actualmente no se utiliza para nada, pero se reserva para un posible uso futuro.

Una vez conocidos todos los pines-hembra de la placa Arduino UNO, es muy interesante observar qué correspondencia existe entre cada uno de ellos y las patillas del microcontrolador ATmega328P. Porque en realidad, la mayoría de estos pines-hembra lo que hacen es simplemente ofrecer de una forma fácil y cómoda una conexión directa a esas patillas, y poco más. Esto se puede ver en la siguiente figura; en ella se muestra cuál es el mapeado de los pines de la placa Arduino respecto a los pines del microcontrolador ATmega328P.



El conector ICSP

Las siglas ICSP (cuyo significado es “In Circuit Serial Programming”) se refieren a un método para programar directamente microcontroladores de tipo AVR, PIC y Parallax Propeller que no tienen el bootloader preinstalado. Ya sabemos que la función de un bootloader es permitir cargar nuestros programas al microcontrolador conectando la placa a nuestro computador mediante un simple cable USB estándar, pero si ese microcontrolador no tiene grabado ningún bootloader, la escritura de su memoria no se puede realizar de esta forma tan sencilla y debemos utilizar otros métodos, como el ICSP.

Esta situación nos la podemos encontrar cuando por ejemplo queramos reemplazar el microcontrolador DIP de una placa Arduino UNO por otro que hayamos adquirido por separado sin bootloader incorporado. En este caso podríamos optar por usar el método ICSP para grabarle un bootloader (para así volver a poderlo programar vía USB directamente; de hecho, así es como se han grabado precisamente los bootloaders en los microcontroladores de las placas Arduino que vienen de fábrica) o también para grabar directamente siempre nuestros programas sin tener que usar ningún bootloader nunca (con la ventaja de disponer entonces de más espacio libre en la memoria Flash del microcontrolador –el que ocuparía el bootloader si estuviera– y de poder ejecutar nuestros programas inmediatamente después de que la placa reciba alimentación eléctrica sin tener que esperar a la ejecución de un bootloader inexistente). Otra situación en la que nos puede interesar utilizar el método ICSP es cuando queramos sobreescribir el bootloader existente por otro, porque el original se haya corrompido, por ejemplo.

Para poder programar un microcontrolador adherido a alguna placa Arduino mediante el método ICSP se necesita un aparato hardware específico, el “programador ISP”. Existen varias formas y modelos, pero hoy en día la versión más extendida de programador ISP es la de un dispositivo que consta por un lado de un conector USB para enchufar a nuestro computador, por otro de una clavija ICSP lista para encajar en el conector ICSP de nuestra placa Arduino, y cuyo “corazón” interno es un determinado microcontrolador especializado en su función de programador.

A pesar de que todos tengan una apariencia extensa similar, hay que tener en cuenta que no todos los programadores ISP son compatibles con todos los modelos de microcontroladores AVR. De entre los que lo son con el microcontrolador ATmega328P (la lista completa se puede consultar en el fichero llamado “programmers.txt”, descargado junto con el entorno de desarrollo de Arduino) podemos encontrar los siguientes:

"AVRISP mkII" (http://www.atmel.com/tools/MATUREAVRISP.aspx): es el programador ISP oficial fabricado por Atmel, basado estrictamente en el protocolo de transferencia STK500 oficial de Atmel. Soporta prácticamente todos los modelos de microcontroladores AVR y los entornos de desarrollo de chips AVR más extendidos (además del entorno Arduino). Existe una versión ya anterior (ya obsoleta) de este programador llamada "AVR ISP".

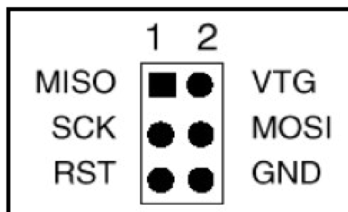
"USBtinyISP" (http://www.ladyada.net/make/usbtinyisp): programador ISP fabricado por Adafruit, que pretende mejorar el anterior. Por ejemplo, una ventaja que tiene respecto el "AVRISP mkII" es que, a diferencia de este, permite alimentar con 5 V/100 mA el microcontrolador a programar (a través de su conexión USB con el computador) sin necesidad de tener que alimentarlo por otro lado. No obstante, no se vende montado sino en forma de kit, por lo que se requiere un mínimo proceso de soldadura de componentes (aunque las instrucciones que se ofrecen son muy claras y detalladas). También se requiere la instalación de un driver si utilizamos el sistema operativo Windows en nuestro computador.

"USBasp" (http://www.fischl.de/usbasp): programador ISP cuyo esquema hardware y firmware es libre. Requiere la instalación de un driver específico en sistemas operativos Windows.

"Arduino as ISP": si tenemos dos placas Arduino y queremos programar el microcontrolador de una de ellas, podemos conectarlas entre sí para que (comunicándose a través de SPI) el microcontrolador de la primera funcione como programador del de la segunda. Si se desean más detalles, en la página http://arduino.cc/en/Tutorial/ArduinoISP viene explicado paso a paso el procedimiento completo.

Otros programadores ISP dignos de mención son el "USB AVR Programmer" de Pololu (producto nº 1300) y el "USB AVR Programmer" de Seeedstudio (producto con código TOL132C1B).

Existen dos estándares para los conectores ICSP, uno de 6 pines y otro de 10, pero este último está obsoleto. La placa Arduino UNO incorpora un conector ICSP de 6 pines cuyo diagrama esquemático se muestra en la figura siguiente.



Cada uno de los pines del conector ICSP está empalmado internamente a una patilla concreta del microcontrolador ATtmega328P. De la figura anterior podemos comprobar que el conector ICSP utiliza en realidad el protocolo estándar SPI para comunicarse con el microcontrolador a programar. Concretamente, tenemos –además del pin de alimentación (VCC) y el de tierra (GND)– el pin de reloj (SCK, –"clock" –, que marca el ritmo al que se transfieren los datos), el pin de salida serie de datos (MISO), el pin de entrada serie de datos (MOSI) y el pin llamado RESET (conectado al pin SS del microcontrolador). La función de este último pin es activar o desactivar la comunicación con el microcontrolador: mientras se reciba un voltaje ALTO no ocurrirá nada, pero cuando se reciba un voltaje BAJO, el ATmega328P detendrá la ejecución del programa que tenga grabado en ese momento y se dispondrá a recibir una reprogramación.

Si hubiéramos adquirido un microcontrolador como pieza suelta y no tuviéramos ninguna placa Arduino con el zócalo DIP libre para colocarlo, para poder programarlo mediante ICSP necesitaríamos un conector ICSP donde enchufar nuestro programador ISP y empalmar cada pin de ese conector con la patilla correspondiente del microcontrolador. Es relativamente sencillo fabricar el circuito pertinente mediante una breadboard. Incluso, si se tienen los conocimientos suficientes, no es difícil fabricarse una pequeña placa auxiliar de tipo perfboard con las conexiones ya establecidas entre conector y microcontrolador. Ejemplos de placas de este tipo (junto con una explicación detallada de su fabricación) se pueden encontrar en el sitio http://www.evilmadscientist.com. También se puede adquirir un modelo exclusivo en Sparkfun, con nº de producto 8508.

Sea como sea, no basta con conectar correctamente nuestro programador ISP para programar nuestro microcontrolador mediante ICSP. Es necesario realizar una serie de pasos que involucran el uso de determinado software ejecutado en nuestro computador. No obstante, el procedimiento concreto no lo podemos explicar todavía porque aún nos faltan los conocimientos necesarios, pero será detallado en el próximo capítulo.

El reloj

Para marcar el ritmo de ejecución de las instrucciones en el microcontrolador, el ritmo de la lectura y escritura de los datos en su(s) memoria(s), el ritmo de adquisición de datos en los pines de entrada, el ritmo de envío de datos hacia los pines de salidas y en general, para controlar la frecuencia de trabajo del microcontrolador, la placa Arduino posee un pequeño "metrónomo" o reloj, el cual funciona a una frecuencia de 16 millones de hercios (16MHz). Esto quiere decir que

(aproximadamente y simplificando mucho) el microprocesador es capaz de realizar 16 millones de instrucciones en cada segundo. Sería posible que la placa Arduino incorporara un reloj con una frecuencia de trabajo mayor: así disminuiría el tiempo en el que se ejecutan las instrucciones, pero esto también implicaría un incremento en el consumo de energía y de calor generado.

Electrónicamente hablando, existen varios tipos de “relojes”. Entre ellos, están los osciladores de cristal y los resonadores cerámicos. Los primeros son circuitos que utilizan un material piezoeléctrico (normalmente cristal de cuarzo, de ahí su nombre) para generar una onda vibratoria de alta frecuencia muy precisa. Los segundos consisten en un material cerámico piezoeléctrico que genera la señal oscilatoria de la frecuencia deseada cuando se le aplica un determinado voltaje.

El reloj que lleva la placa Arduino es un resonador cerámico. Este tipo de relojes son ligeramente menos precisos que los osciladores de cristal, pero son más baratos. En concreto la precisión en la frecuencia aportada por un cristal de cuarzo con compensación de temperatura (los llamados TCXO) es del 0,001% (es decir, que un valor nominal de 16 MHz tendríamos valores como mucho de 16 KHz por arriba o por abajo de este), pero la precisión aportada por un resonador cerámico típico fabricado con PZT (zirconato titanato de plomo) es del 0,5%.

Si no estamos satisfechos con la precisión del reloj que incorpora la placa Arduino, siempre podremos elegir utilizar un componente externo que actúe como reloj diferente del que viene en la propia placa y sincronizar el microcontrolador con él, pero en realidad, para la inmensa mayoría de proyectos que hagamos con Arduino, no nos será necesario recurrir a esto para nada.

Técnicamente, se podría utilizar como reloj del ATmega328P el oscilador de cristal presente en la placa que controla al ATmega16U2, pero en la práctica el circuito resultante generaría demasiado ruido electromagnético.

Por otro lado, no está de más comentar que el microcontrolador ATmega328P en realidad incluye un reloj interno propio (de tipo RC) dentro de su encapsulado, por lo que en teoría no sería necesario utilizar ningún reloj adicional en la placa Arduino. No obstante, ese reloj interno solo es capaz de marcar un “ritmo” de 8 MHz y además, tiene una precisión muy pobre (un 10%) por lo que casi nunca se utiliza.

El botón de “reset”

La placa Arduino UNO dispone de un botón de reinicio (“reset”) que permite, una vez pulsado, enviar una señal LOW al pin “RESET” de la placa para parar y volver a arrancar el microcontrolador. Como en el momento del arranque del microcontrolador siempre se activa la ejecución del bootloader, el botón de reinicio se suele utilizar para permitir la carga de un nuevo programa en la memoria Flash del microcontrolador –eliminando el que estuviera grabado anteriormente– y su posterior puesta en marcha. No obstante, en la placa UNO no es necesario prácticamente nunca pulsar “real y físicamente” dicho botón antes de cada carga, ya que la placa UNO está diseñada de tal manera que permite la activación del bootloader directamente desde el entorno de desarrollo instalado en nuestro computador (simplemente pulsando sobre un icono de dicho entorno).

Esta capacidad que tiene la placa de “resetearse” sin pulsar físicamente ningún botón tiene sus consecuencias: cuando la placa UNO está conectada a un computador ejecutando un sistema operativo Linux o Mac OS X, cada vez que se realiza una conexión a la placa mediante USB, esta se reinicia, por lo que durante aproximadamente medio segundo, el bootloader se pone en marcha. Por tanto, durante ese medio segundo, el bootloader interceptará los primeros bytes de datos enviados a la placa justo tras el establecimiento de la conexión USB. El problema de esto es que, aunque el bootloader no hará nada con estos bytes porque los ignorará, estos ya no llegarán al destino deseado (es decir, el programa grabado en el microcontrolador propiamente dicho), porque este se pondrá en marcha con ese retraso de medio segundo. Por lo tanto, hay que tener muy en cuenta que si nuestro programa grabado en el microcontrolador está pensado para recibir nada más iniciarse datos provenientes de nuestro computador vía USB, nos tendremos que asegurar de que el software encargado de enviarlos se espere un segundo después de abrir la conexión USB para proceder al envío.

Si se desea, se puede deshabilitar la función de “auto-reset”, de tal forma que siempre sea necesario utilizar el pulsador incorporado en la placa para ejecutar el bootloader y, por tanto, cargar un nuevo programa en el microcontrolador. Existen varias maneras para lograrlo, pero la más sencilla es conectar un condensador de 10 microfaradios entre los pines “GND” y “RESET” de la placa. Otra manera más definitiva de quitar el “auto-reset” sería cortar la traza etiquetada como “RESET-EN” en el dorso de la placa; para volver a tener el “auto-reset”, cada extremo de esa traza debería ser soldado de nuevo entre sí. En todo caso, una vez deshabilitado el “auto-reset”, el procedimiento de carga de nuestros programas será el siguiente:

1. Mantener apretado el pulsador de reinicio de la placa.
2. Clicar en el botón "Upload" del entorno de desarrollo Arduino.
3. Tan pronto como se ilumine una vez el LED etiquetado como RX en nuestra placa Arduino, rápidamente soltar el pulsador.
4. La carga debería empezar haciendo parpadear a los LEDs RX y TX.

Obtener el diseño esquemático y de referencia

Los curiosos (con un nivel avanzado de electrónica) que deseen conocer hasta el mínimo detalle cómo está construida la placa Arduino UNO y cómo están interconectados los diferentes componentes que la forman, pueden descargarse el diseño esquemático en formato pdf de la siguiente dirección: http://arduino.cc/en/uploads/Main/Arduino_Uno_Rev3-schematic.pdf

También está disponible el diseño de referencia necesario para construir nosotros mismos una placa de circuito impreso que sea exactamente igual a la de la Arduino oficial (o bien modificada si modificamos estos archivos) y completarla posteriormente añadiendo los componentes necesarios (microcontroladores, pines-hembra, etc.) por separado. Este diseño puede descargarse desde la siguiente dirección: http://arduino.cc/en/uploads/Main/arduino_Uno_Rev3-02-TH.zip. Si descomprimimos este archivo zip, veremos que contiene dos ficheros: el fichero .SCH es un diagrama visual esquemático de la PCB que contiene una serie de símbolos gráficos representando principalmente puertas lógicas y líneas de conexión cuya utilidad es generar fácilmente a partir de él el fichero .BRD, que es el que realmente contiene todos los datos necesarios para la fabricación de la PCB.

El contenido de estos archivos se puede ver y editar mediante un software (no libre) de diseño de PCBs llamado EAGLE –versión 6.0 o posterior–. Se puede descargar de su página oficial http://www.cadsoftusa.com una versión de prueba gratuita de 30 días.

Aclaremos que en estos ficheros de diseño aparece un microcontrolador que no es el que lleva la placa Arduino (en concreto, es el microcontrolador ATmega8), pero no hay ningún problema con esto porque las configuraciones de pines del microcontrolador ATmega8 (y del ATmega168 también) son idénticas a las de los del Atmega328P.

Decir finalmente que la placa Arduino está certificada tanto por la Comisión Federal de Comunicaciones estadounidense (FCC) como por la Unión Europea. Eso significa que Arduino cumple en ambas zonas geográficas con la regulación pertinente en el ámbito de las emisiones electromagnéticas de aparatos electrónicos

y de telecomunicaciones. Podemos comprobar que efectivamente Arduino respeta las normas de ambos organismos observando sus respectivos logos en el dorso de la placa. La importancia de tener estos logos impresos es facilitar la aceptación de las placas Arduino en su acceso a los distintos mercados (estadounidense y europeo según el caso), ya que si no estuvieran certificados, su entrada podría ser rechazada al no cumplir la normativa legal.

¿QUÉ OTRAS PLACAS ARDUINO OFICIALES EXISTEN?

A continuación, haremos un breve resumen de las posibilidades que ofrecen las otras placas Arduino oficiales diferentes del modelo UNO. Todos estas variantes están especializadas en trabajar dentro de circunstancias específicas donde la placa UNO estándar no nos ofrece soluciones a las necesidades que nos puedan surgir. De todas formas, si se desea obtener información más detallada y completa de la que se proporciona en los párrafos siguientes, lo mejor es consultar el enlace siguiente: http://arduino.cc/en/Main/Products, donde se especifican todos los datos técnicos de cada una de estas placas.

Arduino Mega 2560

Placa basada en el microcontrolador ATmega2560. Como características más destacables diremos que tiene 54 pines de entrada/salida digitales (de los cuales 14 pueden ser usados como salidas analógicas PWM), 16 entradas analógicas y 4 receptores/transmisores serie TTL-UART. Consta de una memoria Flash de 256 Kilobytes (de los cuales 8 están reservados para el bootloader), una memoria SRAM de 8 KB y una EEPROM de 4 KB. Su voltaje de trabajo es igual al del modelo UNO: 5 V.

En la sección correspondiente a esta placa dentro de la web oficial de Arduino podemos descargarnos los ficheros del diseño esquemático de la placa en PDF, los ficheros del diseño de la PCB en el formato propio del programa EAGLE y una imagen ilustrativa del mapeado de los pines del microcontrolador con relación a los pines de la placa. También nos podemos descargar la documentación oficial del microcontrolador ATmega2560.

Arduino Mega ADK

Placa muy similar a la Mega 2560. La diferencia principal está en que la Mega ADK es capaz de funcionar como un dispositivo de tipo “host USB” (y la Mega 2560 no). Expliquemos esto.

En una comunicación USB entre diferentes dispositivos siempre existe uno que actúa como "maestro" (el llamado "host") y el otro –u otros– que actúan como "esclavos" (los llamados "periféricos"). El "host" es el único que puede iniciar y controlar la transferencia de datos entre el resto de dispositivos conectados, mientras que los "periféricos" tan solo pueden responder a las peticiones hechas por el "host" y poca cosa más. Los dispositivos "host" disponen de un conector USB de tipo A, y los dispositivos "periféricos" disponen de un conector USB de tipo B, mini-B o micro-B. Un "host" típico es un computador, al cual se le pueden conectar varios "periféricos", como lápices de memoria, cámaras de fotos o video, teléfonos móviles de última generación, etc.

La placa Arduino ADK puede funcionar como periférico USB igual que el resto de placas Arduino (por eso incorpora el conector USB de tipo B como las demás) pero también como host USB (y por eso, como novedad, también incorpora un conector USB de tipo A). Pero no es suficiente con tener el conector adecuado para que la placa pueda ser un host USB: en realidad este modo de funcionamiento es posible gracias a que la placa incorpora un chip específico (un microcontrolador, de hecho) que implementa la lógica necesaria: el MAX3421E del fabricante Maxim, el cual se comunica con el ATmega2560 mediante SPI.

Así pues, a la placa ADK se puede conectar cualquier dispositivo que tenga un puerto USB periférico (teléfonos móviles, cámaras de fotos o vídeo, teclados, ratones, joysticks y mandos de diferentes videoconsolas, etc.) para controlarlo e interactuar directamente con él.

En concreto, la placa Arduino ADK está especialmente diseñada para interactuar con teléfonos móviles funcionando con el sistema Android (http://www.android.com), desarrollado por Google. La idea es que se puedan escribir programas para Android que se relacionen con el código Arduino ejecutado en ese momento en la placa, de tal forma que se establezca una comunicación entre el móvil y la placa que permita, por ejemplo, realizar un control remoto desde el dispositivo Android de los sensores y/o actuadores conectados al hardware Arduino.

La combinación de Arduino con Android es un tema tremendamente interesante, pero muy extenso y relativamente complejo (se necesita saber cómo funciona internamente la plataforma de Google, y en concreto, el kit de desarrollo "Accesory Dev Kit"), por lo que en este libro no se abordará. Si se desea conocer más, se puede consultar la bibliografía aportada final del libro o empezar por aquí: http://labs.arduino.cc/ADK/Index . También se puede consultar la documentación oficial de Android, y en especial los apartados correspondientes a la comunicación vía

USB: http://developer.android.com/guide/topics/usb/adk.html y también http://developer.android.com/guide/topics/usb/index.html

Arduino Ethernet

Al igual que el modelo UNO, la placa Ethernet está basada en el microcontrolador ATmega328P (y por lo tanto, tiene la misma cantidad de memoria Flash, SRAM y EPPROM), y también tiene el mismo número de pines de entrada/salida digitales y de entradas analógicas. El resto de características también es muy similar al modelo UNO. La mayor diferencia que existe con la placa UNO es que la placa Ethernet incorpora un zócalo de tipo RJ-45 para poder conectarse mediante el cable adecuado (cable de par trenzado de categoría 5 o 6) a una red de tipo Ethernet.

Las redes Ethernet, basadas en la especificación IEEE802.3 (http://www.ieee802.org/3), son la tecnología más extendida con diferencia en la construcción de redes de área local (las llamadas "LAN" –Local Area Network–). Las redes LAN permiten el tráfico de datos entre los distintos dispositivos conectados a ella (computadores, impresoras, "routers", etc.) dentro de una zona de distancias relativamente reducidas, tal como un edificio o un conjunto de edificios adyacentes. Aunque técnicamente no tenga por qué, en la práctica las redes Ethernet utilizan el mismo conjunto de protocolos de comunicación que Internet (la llamada pila "TCP/IP").

La placa Arduino Ethernet permite, pues, transferir datos entre ella misma (los cuales pueden ser obtenidos de algún sensor, por ejemplo) y cualquier otro dispositivo conectado a su misma red LAN (normalmente, un computador que los recopila y guarda), o viceversa: transferir datos entre un dispositivo conectado a la LAN (normalmente, un computador ejecutando algún tipo de software de control) y ella misma (la cual puede estar conectada a algún actuador controlado remotamente por ese computador). También se puede lograr, gracias al establecimiento del enrutamiento de paquetes adecuado, comunicar nuestra placa Ethernet con cualquier dispositivo conectado a cualquier red del mundo fuera de nuestra LAN privada (incluyendo "Internet"), con lo que sus posibilidades de uso se disparan.

Evidentemente, solo con disponer de un zócalo RJ-45 la placa Arduino Ethernet no es capaz "por arte de magia" de comunicarse dentro de una red. Lo que permite que esto ocurra es la inclusión dentro de la placa de un chip controlador de Ethernet que, de hecho, implementa por hardware toda la pila de protocolos TCP/IP (en concreto los protocolos TCP, UDP, ICMP, Ipv4, ARP, IGMPv2, PPPoE y por

supuesto Ethernet). Se trata del W5100 del fabricante Wiznet. Este chip es pues el verdadero responsable de que la placa Arduino Ethernet pueda manejar redes de este tipo.

Que este chip "implemente por hardware la pila TCP/IP" significa en pocas palabras que permite al desarrollador de programas Arduino utilizar de forma muy sencilla la red (mediante la librería de programación "Ethernet", incorporada por defecto dentro del lenguaje Arduino) para poder transmitir o recibir datos sin tener que preocuparnos por detalles más técnicos (como el control de errores en la transmisión de datos, la sincronización de las señales y datagramas, etc.) de los cuales ya se encarga directamente este chip. Sobre el uso detallado de la librería de programación "Ethernet" (incluyendo los conceptos básicos de redes como la configuración de direcciones IP, direcciones MAC, etc.) y sus posibles aplicaciones hablaremos en el capítulo 8.

Entre otras características, conviene saber que el W5100 puede funcionar a velocidades de transmisión de 10 y 100 megabits por segundo, que permite hasta cuatro conexiones simultáneas independientes y que dispone de una memoria interna de 16 kilobytes para almacenar temporalmente datos enviados o recibidos de la red. El W5100 se comunica mediante el protocolo SPI con el ATmega328P, por lo que hay que tener en cuenta que los pines digitales números 10, 11, 12 y 13 de la placa Arduino Ethernet están reservados (es decir, no se pueden utilizar para otra cosa). Por esta razón, el pin que está asociado al LED que viene incorporado en la placa no es el número 13 como ocurría con la placa UNO, sino que es el número 9, y además se reduce el número de pines disponibles respecto el modelo UNO a 9, con solo cuatro pines disponibles como salidas PWM.

La placa Arduino Ethernet también dispone de un zócalo para insertar una tarjeta microSD, la cual puede ser usada (mediante la librería de programación "SD", incorporada por defecto dentro del lenguaje Arduino) para guardar diferentes tipos de ficheros y ofrecerlos a través de la red (que recordemos, puede ser nuestra LAN privada o, si queremos, el mundo entero). Hay que tener en cuenta que si la tarjeta microSD está presente, el pin 4 está reservado para el control de esta.

Los adaptadores USB-Serie

Lo que tal vez llame más la atención de la placa Arduino Ethernet es que no dispone de zócalo USB de ningún tipo (es el "precio a pagar" de disponer en cambio de un zócalo RJ-45). Esto quiere decir que desde nuestro computador no podemos comunicarnos mediante USB (que es la forma "estándar") con el bootloader de

nuestro microcontrolador para poderle grabar en su memoria Flash el programa que deseemos que ejecute. Para resolver este obstáculo, podríamos utilizar un programador ISP, pero lo más sencillo es adquirir y utilizar el adaptador USB-Serie oficial (http://arduino.cc/en/Main/USBSerial). Este adaptador es simplemente una plaquita que contiene un zócalo USB de tipo mini-B y el microcontrolador ATmega8U2, un chip muy similar al ya conocido ATmega16U2 (pero con la mitad de memoria Flash, de ahí su nombre) que está programado para realizar exactamente la misma función: convertir la conexión USB en una señal serie simple de 5 V entendible por el microcontrolador ATmega328P de la placa Ethernet.

Por tanto, para poder conectarnos vía USB a la placa Ethernet deberemos enchufar primero este adaptador a los seis pines que sobresalen de la placa Ethernet (cada uno de los cuales está conectado a su vez directamente a una determinada patilla del ATmega328P: RX, TX, reinicio, alimentación, tierra...) y seguidamente conectar el cable USB al zócalo ofrecido por el adaptador. Al igual que haríamos con la placa UNO, el conector USB recién añadido lo podremos utilizar a partir de entonces tanto para alimentar la placa eléctricamente como para programarla (incluso con el auto-reset incorporado también).

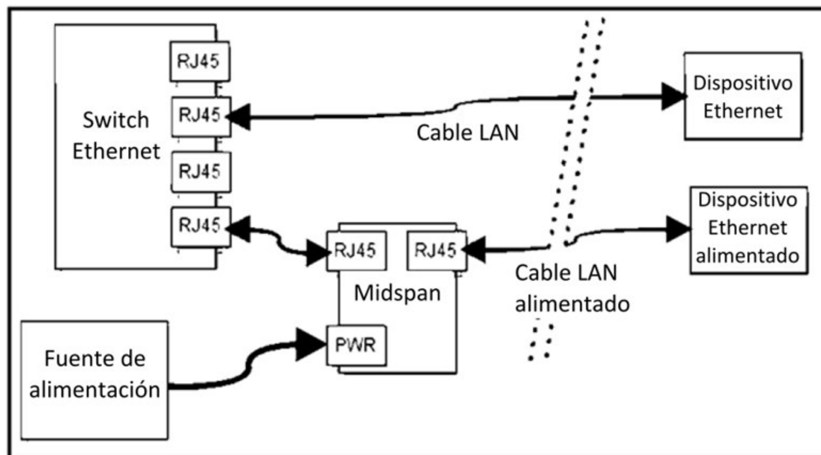
No obstante, si no se desea utilizar el adaptador oficial, existen muchos otros adaptadores que realizan la misma función, aunque en general incorporan un chip conversor USB-Serie diferente: el FT232RL del fabricante FTDI. Un ejemplo de estos son las placas adaptadoras llamadas "FTDI Basic Breakout" distribuidas por Sparkfun (las cuales según el modelo pueden funcionar a 5 V o 3,3 V) o también los cables adaptadores "USBTTLSerial" oficiales de FTDI (concretamente los modelos TTL-232R-5V y TTL-232R-3V3, los cuales, tal como indica su nombre, funcionan a 5 V o 3,3 V, respectivamente). Placas similares son el "FTDIFriend" de Adafruit , el "FTDI Adapter" de Akafugu o el "Foca" de IteadStudio, entre muchas otras.

PoE ("Power Over Ethernet")

Algo que conviene conocer en relación con la placa Arduino Ethernet es que existe la posibilidad de alimentarla eléctricamente a través del propio cable Ethernet, sin usar ni cable USB ni fuente de alimentación externa (adaptador AC/DC, pila, etc.). Es decir, se puede aprovechar la conexión de datos que ofrece el cable de par trenzado típico de una red Ethernet para recibir por allí también el voltaje necesario para el correcto funcionamiento de la placa, sin necesidad, pues, de utilizar ningún otro cable. Esto es muy conveniente en instalaciones que por su tamaño o su ubicación hagan complicado el añadido de varios cables.

Para ello, no obstante, se han de cumplir diferentes condiciones. La primera es que a través del cable Ethernet viaje la señal adecuadamente transformada. Esto se puede conseguir de dos maneras diferentes:

Utilizando un inyector "midspan". Este es un dispositivo que se enchufa por un lado tanto a la alimentación eléctrica externa (mediante un adaptador AC/DC, generalmente) como a un switch de red estándar (mediante un cable Ethernet), y por otro con el dispositivo a alimentar (mediante solo otro cable Ethernet). Los datos atravesarán el inyector desde/hacia el switch o desde/hacia el dispositivo final por los cables Ethernet de ambos tramos sin alteraciones, pero además, la alimentación eléctrica recibida por el inyector será transmitida por el cable Ethernet conectado al dispositivo final. Básicamente el esquema descrito sería la siguiente figura:

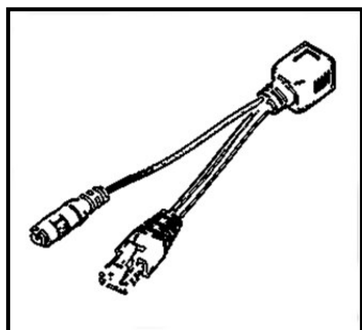


Utilizando un switch PoE. Este es un switch (un conmutador de red) especial que además de funcionar como un switch Ethernet estándar (intercomunicando entre sí los dispositivos conectados a él), tiene la capacidad de proporcionar alimentación eléctrica a través de sus zócalos RJ-45. Por tanto, los dispositivos que queramos que tanto estén conectados entre sí vía Ethernet como que estén alimentados eléctricamente vía PoE debemos enchufarlos a este tipo de switch, mediante un cable Ethernet estándar cualquiera.

Desgraciadamente, los primeros sistemas PoE (tanto inyectores midspan como switches PoE) fueron creados por varias empresas sin seguir ningún estándar concreto, por lo que existen diferentes sistemas PoE incompatibles entre sí (ya que

usan diferentes voltajes, diferentes polaridades, distintas disposiciones de pines, etc.). Afortunadamente, una especificación estándar fue definida finalmente y los sistemas PoE actuales han convergido hasta llegar a un punto donde son casi compatibles. Esta especificación se llama “802.3af” (del 2003) o, la más moderna, “802.3at” (del 2009).

No obstante, estos estándares son demasiado difíciles de implementar porque estipulan un valor de voltaje a usar de hasta 48 V (mucho más del voltaje que los componentes usados en nuestros proyectos pueden manejar, incluso para los reguladores de tensión) y un esquema de señalizaciones relativamente complejo que el dispositivo a alimentar ha de implementar para informar al inyector sobre la cantidad de tensión que requiere. Por ello, muchas veces se suele utilizar un sistema simplificado que utiliza las mismas conexiones que el estándar 802.3 pero que funciona a una tensión más reducida, y sin esquema de señalizaciones.



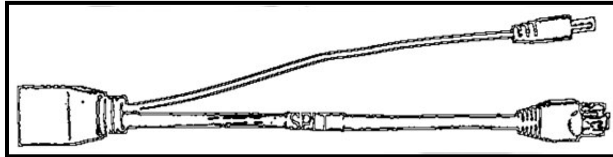
En otras palabras: para poder utilizar PoE con Arduino no es necesario utilizar un switch o un inyector “midspan” que soporte el estándar 802.3 al completo, los cuales resultan ser además bastante caros. Podemos utilizar simplemente un inyector “midspan” tan elemental como el llamado “Passive PoE Cable Set” de Sparkfun (mostrado en la ilustración de la izquierda), el cual consta simplemente por un lado de un conector “jack” de 2,1 mm hembra para empalmar con la fuente de alimentación y un conector RJ-45 para enchufar en un switch, y por el otro tiene un zócalo RJ-45 donde se colocará el cable Ethernet que llevará la señal PoE al dispositivo alimentar. Sparkfun lo vende junto con un splitter PoE (enseguida veremos qué es esto) con código de producto 10759.

Otro inyector “midspan” algo más sofisticado pero también muy barato y sencillo de usar es el llamado “4-Channel PoE Midspan Injector” fabricado por Freetronics, el cual ofrece un voltaje de hasta 24 V (de sobra para lo que una placa Arduino es capaz de manejar). Concretamente, este inyector consta de un conector de alimentación “jack” de 2,1 mm preparado para ser conectado a una fuente de alimentación que proporcione un voltaje de entre 7 V y 24 V (AC o DC), 4 zócalos RJ-45 pensados para ser conectados a un switch de red estándar y 4 zócalos RJ-45 pensados para ser conectados a distintos dispositivos a alimentar. Esta alimentación se realizará gracias a un voltaje de salida del inyector que será de idéntico valor al de entrada pero siempre DC. Por lo tanto, según el valor de la tensión aplicada al

inyector, es posible que se necesite acoplar entre este y el dispositivo un regulador de tensión que la rebaje hasta valores más aceptables para nuestras placas Arduino.

La segunda condición que se ha de cumplir para poder utilizar PoE es que el dispositivo alimentado (y conectado a la vez) por el cable Ethernet disponga de la capacidad de procesar correctamente la señal PoE que le llegue por dicho cable. Si ese dispositivo ya de por sí está habilitado para manejar señales PoE, no hay más que hacer: simplemente se le conecta el cable Ethernet y listo. Si ese dispositivo, en cambio, no es capaz de interpretar las señales PoE, tan solo entenderá la comunicación de datos pero no podrá recibir alimentación eléctrica. Para conseguir esto, se puede hacer de varias maneras:

Utilizar un "splitter PoE". Este no es más que un cable que por un lado tiene un zócalo RJ-45 donde se ha de insertar el cable Ethernet que transmite la señal PoE y que por el otro se divide en dos conectores a enchufar al dispositivo: un conector RJ-45 que transmite los datos y un conector "jack" de 2,1 mm, que transmite la corriente continua. La siguiente figura muestra un ejemplo de "splitter PoE" .



Acoplar una pequeña plaquita extra (también llamado "módulo") al dispositivo en cuestión, que dote a este de la capacidad de procesar la señal PoE convenientemente recibida a través del cable Ethernet. En el caso concreto de la placa Arduino Ethernet, se puede adquirir el módulo PoE Ag9120-S, específicamente adaptado a ella, el cual ofrece 9 V de salida a la placa con un rango de voltaje de entrada por el cable LAN de entre 36 V y 48 V. No obstante, este módulo no es fabricado oficialmente por el Arduino Team porque es hardware privativo. Se puede adquirir bien en las webs de los distintos distribuidores listados en el apéndice A, o bien en la web del propio fabricante *(*http://www.silvertel.com/poe_products.htm*).*

Arduino Fio

Esta placa contiene un ATmega328P funcionando a 3,3 V y a 8 MHz. Tiene 14 agujeros que pueden utilizarse (mediante soldadura directa o bien mediante la colocación de pines-hembra de plástico) como pines de entrada/salida digitales (6 de

los cuales pueden ser usados como salida PWM); también tiene 8 agujeros preparados para utilizarse como entradas analógicas y un botón de reinicio, todo ello dentro de un tamaño muy reducido.

Una novedad de esta placa respecto a las anteriores es que se puede alimentar eléctricamente mediante una batería LiPo gracias a que la placa dispone de un zócalo de tipo JST de 2 pines para poder conectarla directamente allí. La placa Arduino Fio permite ser alimentada también mediante conexión USB, ya que dispone de un conector USB mini-B para ello. De hecho, a través de la alimentación recibida vía USB incluso se puede recargar la batería LiPo que esté conectada en ese momento, ya que la placa incorpora el chip cargador MAX1555 del fabricante Maxim. No obstante, la conexión USB no está pensada para programar el microcontrolador, por lo que se requiere para ello acoplar un adaptador USB-Serie (como los ya comentados cuando vimos la placa Arduino Ethernet) a los agujeros de la placa marcados como GND, AREF, 3V3, RXI, TXO y DTR, mediante la ristra de pines adecuada.

De todas formas, la novedad más interesante de esta placa es la posibilidad de colocarle un módulo XBee en el zócalo que incorpora específico para ello. "XBee" es el nombre comercial dado por el fabricante Digi International a una familia de emisores/receptores de señales de radiofrecuencia de bajo consumo y con un encapsulamiento y tamaño compatible entre sí. Así pues, la placa Arduino Fio está pensada para aplicaciones inalámbricas que sean autónomas en su funcionamiento y que no requieran por tanto un alto nivel de mantenimiento. Un caso muy habitual es conectar a esta placa algún sensor de cualquier tipo junto con el módulo XBee para crear con varias placas Fio una red inalámbrica que permita interrelacionar estos sensores entre sí y con algún computador central recopilador de datos. Los módulos XBee, no obstante, no los estudiaremos en este libro.

Arduino Pro

Esta placa viene en dos "versiones": ambas contienen un microcontrolador Atmega328P SMD, pero una funciona con 3,3 V y a 8 MHz y la otra funciona con 5 V y a 16 MHz. Dispone de 14 agujeros pensados para funcionar como pines de entrada/salida digital (6 de los cuales pueden ser usados como salida PWM), 6 agujeros para entradas analógicas, agujeros para montar un conector de alimentación de 2,1 mm, un zócalo JST para una batería LiPo externa, un interruptor de corriente, un botón de reinicio, un conector ICSP y los pines necesarios para conectar un adaptador o cable USB-Serial y así poder programarla (y también alimentarla) directamente vía USB.

Esta placa está pensada para instalarse de forma semi-permanente en objetos o exhibiciones. Por eso no viene con los pines montados sino que hay colocar en los agujeros los pines-hembra de plástico “a mano” (o bien soldar los cables directamente). De esta manera, se permite el uso de diferentes tipos de configuraciones según las necesidades.

Arduino Lilypad

La placa Arduino LilyPad está diseñada para ser cosida a material textil. Permite además conectarle (mediante hilos conductores) fuentes de alimentación, sensores y actuadores de forma que se puedan “llevar encima”, haciendo posible la creación de vestidos y ropa “inteligente”. Además, se puede lavar. Esta placa incorpora el microcontrolador ATmega328V (una versión de bajo consumo del Atmega328P), el cual se programa acoplando a la placa un adaptador o cable USB-Serie.

En el enlace http://www.sparkfun.com/categories/135 se pueden encontrar bastantes complementos interesantes adaptados en tamaño y flexibilidad al Arduino LilyPad, como por ejemplo sensores de temperatura, luz y acelerómetros, LEDs de diferentes colores, motores de vibración, zumbadores, transmisores-receptores inalámbricos, portapilas para baterías LiPo, botón o AAA, breadboards, interruptores, botones, variantes específicas de la placa, etc. Incluso se distribuyen conjuntos de estos complementos en kits llamados “ProtoSnaps” para una mayor comodidad.

Por otro lado, en la tienda oficial de Arduino (ver apéndice A) se puede adquirir el “Wearable Kit”, formado por un conjunto de componentes (resistencias, potenciómetros, breadboards, hilo conductor…), un sensor de presión y actuadores varios (botones, LEDs…), todos ellos textiles. Este kit está diseñado por la empresa especializada Plug&Wear.

Para saber más sobre posibles usos y trucos de esta placa, se pueden consultar los tutoriales nº 281, nº 308, nº 312, nº 313 y nº 333 de Sparkfun, y también la muy completa y práctica web de su co-diseñador, Leah Buechley http://web.media.mit.edu/~leah/LilyPad. Si se desean conocer proyectos ya realizados con estos componentes, un buen lugar para inspirarse es http://www.kobakant.at/DIY

Arduino Nano

La característica más destacable de esta placa es que a pesar de su tamaño (0,73 pulgadas de anchura por 1,70 de longitud), sigue ofreciendo el mismo número

de salidas y entradas digitales y analógicas que la Arduino UNO y la misma funcionalidad que esta. La consecuencia más evidente de su reducido tamaño es que carece del conector de alimentación de 2,1 mm (aunque puede seguir siendo alimentada por una fuente externa mediante el pin "Vin" o "5 V") e incorpora un conector USB mini-B en vez del conector USB tipo B.

Otra diferencia es que, aunque la placa Arduino Nano se sigue basando en el microcontrolador ATmega328P (en formato SMD), el conversor USB-Serie que lleva incorporado es el chip FTDI FT232RL y no el ATmega16U2.

Esta placa está especialmente pensada para conectarla a una breadboard mediante las patillas que sobresalen de su parte posterior, pudiendo formar parte así de un circuito complejo de una manera relativamente fija.

Arduino Mini

Esta placa es muy parecida a la placa Arduino Nano: está basada igualmente en el microcontrolador ATmega328P SMD funcionando a 16MHz, tiene 14 pines de entrada/salida digitales (6 de los cuales pueden funcionar como salidas PWM) y 8 entradas analógicas. Y al igual que la placa Arduino Nano, la Arduino Mini está especialmente pensada para conectarla a una breadboard mediante las patillas que sobresalen de su parte posterior, pudiendo formar parte así de un circuito complejo de una manera relativamente fija.

La diferencia más importante con la placa Arduino Nano está en que la Arduino Mini (para ahorrar aún más espacio físico y así conseguir un tamaño realmente mínimo de 0,7 pulgadas de ancho por 1,3 de largo) no incorpora ningún chip conversor USB-Serie. Debido a ello, para su programación se necesita utilizar un adaptador USB-Serial externo. En concreto, se recomienda el uso de uno específico y oficial: el llamado "Mini USB", basado en el chip FTDI FT232RL (http://arduino.cc/en/Main/MiniUSB), el cual se ha de colocar por separado en la breadboard y entonces establecer las conexiones pertinentes entre este y el Arduino Mini mediante cables. Para saber todos los detalles concretos de esta configuración, y en general, conocer diferentes posibles usos de esta placa, recomiendo consultar la guía oficial: http://arduino.cc/en/Guide/ArduinoMini.

Arduino Pro Mini

Esta placa tiene el mismo tamaño que una placa Arduino Mini, y una disposición compatible de pines. Viene en dos "versiones": ambas contienen un

microcontrolador ATmega168 pero una funciona con 3,3 V y a 8 MHz y la otra funciona con 5 V y a 16 MHz. También incorpora un botón de reinicio y los pines necesarios para conectar un adaptador o cable USB-Serie y así poder programarla (y también alimentarla) directamente vía USB. También se puede alimentar eléctricamente mediante una fuente externa conectada al pin "Vcc".

Esta placa está pensada para instalarse de forma semi-permanente en objetos o exhibiciones. Por eso no viene con los pines montados sino que hay colocar en los agujeros los pines-hembra de plástico "a mano" (o bien soldar cables directamente). De esta manera, se permite el uso de diferentes tipos de configuraciones según las necesidades.

Arduino Leonardo

La gran novedad de esta placa es que el microcontrolador que incorpora es el ATmega32U4 (en formato SMD), el cual tiene todas las funcionalidades que ofrece el Atmega328P pero incorpora además 0,5 kilobytes más de memoria SRAM y sobretodo, soporta comunicaciones USB directamente (y por tanto, no necesita de ningún chip suplementario como el ATmega16U2 o el FTDI).

Otras diferencias con la placa UNO es que la placa Leonardo incorpora un pin-hembra digital más que la UNO para ser usado como salida PWM (el nº 13) y 6 entradas analógicas extra más, las cuales están situadas físicamente en los pines-hembras digitales marcados con un puntito en el exterior de la placa.

Otra diferencia es que los pines SDA y SCL para la comunicación I^2C/TWI cambian de ubicación respecto la UNO y ahora pasan a estar en los pines-hembra digitales nº 2 y nº 3. Por otro lado, en la placa Leonardo desaparecen los pines GPIO SPI, por lo que la única manera de comunicar esta placa con el exterior mediante este protocolo es utilizando directamente los pines ICSP.

El hecho de que la placa Leonardo solo incluya un microcontrolador tanto para ejecutar los programas como para comunicarse directamente a través de USB con el computador permite que esta placa pueda simular fácilmente (si se programa convenientemente) ser un teclado o un ratón USB conectados a dicho computador. Técnicamente, cuando la placa Leonardo se conecte con un cable USB al computador, este detectará dos "puertos" de conexión diferentes: un puerto USB estándar listo para usar la placa Leonardo como un periférico USB más (lo típico sería un teclado o un ratón, tal como hemos dicho) y otro puerto diferente, similar al generado cuando se conecta la Arduino UNO, usable de la forma "tradicional", para

la programación y comunicación con la placa a través del entorno de programación Arduino.

El "auto-reset" de la placa Leonardo

Debido también a que la placa Leonardo utiliza un único microcontrolador tanto para la ejecución de los programas como para la comunicación USB con el computador, al reiniciar el microcontrolador (apretando físicamente el botón de "reset" de la placa o bien a través del botón correspondiente del entorno de desarrollo) la conexión USB similar a la UNO (la otra no) se interrumpe y se vuelve a restablecer. Esto no ocurre con la placa UNO porque allí la conexión USB la mantiene siempre un chip independiente (el ATmega16U2) que nunca es reiniciado. Una consecuencia de este hecho es que cualquier programa que esté comunicándose en ese momento mediante una conexión serie a través de USB con la placa Leonardo perderá su conexión.

Otra consecuencia es que la carga de los programas realizada por el bootloader se demorará unos cuantos segundos (generalmente, unos ocho) ya que después de haber activado el reinicio, el entorno de desarrollo Arduino deberá esperar hasta que detecte otra vez una conexión USB activa y efectuar entonces la carga del programa a través de ella. Si se utiliza el botón de "reset" físico, esta demora implica que deberemos mantener pulsado este botón durante esos segundos hasta que veamos el mensaje "Uploading…" en la barra inferior del entorno de desarrollo: no lo podremos soltar antes.

Técnicamente, el "auto-reset" de la placa Leonardo se activa cuando el computador envía una señal serie de 1200 bits/s y la cierra. Esto quiere decir que en el arranque de la placa no habrá demora en la ejecución del programa grabado, ya que si no se recibe esa señal, el bootloader no se ejecutará.

Arduino Micro

Esta placa ofrece las mismas funcionalidades que la Arduino Leonardo (tiene por ejemplo el mismo microcontrolador ATmega32U4 a 16MHz, los mismos 32KB de memoria Flash y 2,5 KB de memoria SRAM, el mismo bootloader, el mismo voltaje de trabajo –5 V–…) pero con tamaño realmente mínimo: 48 x 18 mm, ideal para ser ubicado sobre una breadboard sin ocupar apenas espacio. Al igual que el modelo Leonardo, se puede programar a través de una conexión USB (dispone de un zócalo mini-B para ello), pudiendo funcionar además como teclado o ratón simulado. Las características del "auto-reset" de la placa Leonardo también son aplicables a la

Micro. Puede ser alimentada a través del cable USB o bien mediante una fuente de alimentación externa conectada a los pines "Vin" y "GND".

Arduino Due

Esta placa pertenece a una familia totalmente distinta de la del resto de placas Arduino. Incluye el microcontrolador SAM3X8E, el cual, aunque fabricado también por Atmel, es de una arquitectura interna muy diferente a la AVR (concretamente es de tipo ARM Cortex-M3) y además, sus registros son cuatro veces más grandes de lo habitual en las otras placas (concretamente, son de 32 bits). Su velocidad de reloj está también muy por encima del resto de placas Arduino (concretamente, es de 84 MHz). Además, el microcontrolador SAM3X8E dispone de muchas más memoria (concretamente, 96 KB de SRAM y 512 KB de memoria Flash) y también de un circuito especializado (llamado controlador "DMA") que permite a la CPU acceder a la memoria de una manera mucho más rápida.

Todo esto implica que con la placa Arduino Due se pueden hacer más cosas, y más rápidamente, por lo que permite ejecutar aplicaciones que realizan un gran procesado de datos. El precio de venta también es mayor que el resto de placas Arduino, lógicamente.

Otros datos técnicos de la Arduino Due son: dispone de 54 pines de entrada/salida digital (12 de los cuales pueden ser usados como salidas PWM), 12 entradas analógicas, 4 chips TTL-UART (es decir, cuatro canales serie hardware independientes), 2 conversores digitales-analógicos (¡novedad!), 2 puertos I^2C independientes, 1 puerto SPI (el cual solo está implementado en los pines "ICSP"), 1 conector USB de tipo mini-B, 1 conector USB de tipo mini-A, un zócalo de 2,1 mm tipo "jack", un botón de reinicio y un botón de borrado. Además, ofrece como es habitual los pines "Vin", "GND", "5 V" y "3,3 V".

Un aspecto muy importante de esta placa que hay que saber es que su voltaje de trabajo es de 3,3 V. Esto significa que la tensión máxima que los pines de entrada/salida pueden soportar es ese. Si se les proporciona un voltaje mayor (como los 5 V a los que estamos habituados), la placa se podría dañar. No obstante, las fuentes de alimentación externas pueden ser las mismas que las utilizadas con la placa Arduino UNO, ya que sus rangos de tensión de entrada son idénticos (6-20 V teóricos, 7-12 V recomendables); esto es debido a que la tensión es adecuadamente reducida gracias a un regulador interno. Por otro lado, la intensidad ofrecida por los pines de salida está entre 6 mA y 15 mA, y la ofrecida por los pines "3,3" y "5 V" es de 800 mA.

La Arduino Due mantiene la forma y disposición de la placa Arduino Mega, siendo compatible con todos los shields que respeten la misma disposición de pines y que, importante, trabajen a 3,3 V. Por otro lado, todos los pines de entrada/salida tienen una resistencia “pull-up” interna desconectada por defecto de 100 KΩ.

La Arduino Due ofrece dos conectores USB para separar dos funcionalidades diferentes. El conector más cercano al jack de alimentación (mini-B) está pensado para enchufar la placa al computador y transferir desde el entorno de desarrollo nuestro programa para que sea ejecutado por el microcontrolador, y a partir de allí mantener la comunicación serie entre computador y placa. De hecho, este conector está controlado por el mismo chip ATmega16U2 que la placa Arduino UNO, por lo que su comportamiento es idéntico. El conector más cercano al botón de reinicio (mini-A), en cambio, está controlado directamente por el chip SAM3X8E y está pensado para usar la placa como un periférico USB más (como un teclado o un ratón, tal como también ocurre en la placa Leonardo). Pero además, una novedad que ofrece este último conector es que también permite a la placa actuar como “host USB” (tal como lo hace la placa Mega). De esta forma, no solamente podríamos usar la placa Due como un teclado o un ratón, sino que podríamos conectarle a ella un teclado o un ratón reales, entre otros muchos dispositivos, como teléfonos móviles de última generación, por ejemplo.

La forma de programar la placa Arduino Due es similar a las anteriores placas, tanto en el uso del entorno de desarrollo como en el propio lenguaje de programación: todos los cambios están “bajo la superficie”. De todas formas, la versión actual del entorno de desarrollo a fecha de edición del libro (la 1.0.2) no permite todavía el uso de esta placa, por lo que es necesario descargar la versión 1.5, la cual en el momento de la edición de este libro, está en estado beta. La descarga se puede realizar desde aquí http://arduino.cc/en/Main/SoftwareDue .

El único detalle que hay que tener en cuenta es que la memoria Flash del microcontrolador ha de ser borrada “manualmente” cada vez que se le quiera cargar en nuestro programa. Esto es así porque el bootloader de esta placa está alojado en una memoria de tipo ROM separada de la memoria Flash, y solamente se ejecuta cuando detecta que la memoria Flash está vacía. De ahí la existencia del botón de borrado (marcado como “Erase”) en la placa. Afortunadamente, si conectamos esta a nuestro computador mediante el zócalo USB mini-B (el más cercano al jack de alimentación), este proceso de borrado se realiza de forma automática. En este sentido, la comunicación USB entre placa y computador se realiza de la misma forma ya sea utilizando la placa Due (mediante el conector mini-B) como la placa UNO.

¿QUÉ “SHIELDS” ARDUINO OFICIALES EXISTEN?

Además de las placas Arduino propiamente dichas, también existen los llamados “shields”. Un “shield” (en inglés significa “escudo”) no es más que una placa de circuito impreso que se coloca en la parte superior de una placa Arduino y se conecta a ella mediante el acoplamiento de sus pines sin necesidad de ningún cable. Su función es actuar como placas supletorias, ampliando las capacidades y complementando la funcionalidad de la placa Arduino base de una forma más compacta y estable.

Dependiendo del modelo, incluso se pueden apilar varios shields uno encima de otro. Esto dependerá de si el shield inferior ofrece pines-hembra para poder acoplarlos a los pines sobresalientes del dorso del shield superior.

Normalmente, los shields comparten las líneas GND, 5 V (o 3V3), RESET y AREF con la placa Arduino, y además suelen monopolizar el uso de algunos pines de entrada/salida para su propia comunicación con ella, por lo que estos quedan “inutilizados” para cualquier otro uso. Por ejemplo, si se acoplan varios shields uno sobre otro y todos se comunican mediante SPI con la placa Arduino, todos ellos podrán utilizar sin problemas los pines comunes correspondientes a las líneas MISO, MOSI y SCLK , pero cada una de ellos deberá usar un pin diferente como línea CS.

Por otro lado, también hay que tener en cuenta los requerimientos de alimentación eléctrica que necesitan los shields. Ya sabemos que una placa Arduino recibe alrededor de 500 mA (ya sea mediante conexión USB o mediante conexión jack externa), por lo que la corriente que queda para el funcionamiento de un posible shields es pequeña. Ejemplos de shields que consumen mucho (de hasta 300 mA) son los que tienen pantallas LCD o los que proporcionan conectividad Wi-Fi. También hay que tener en cuenta si un shield determinado necesita una tensión de 3,3 V.

Existen literalmente centenares de shields construidos por la comunidad compatibles con la placa UNO que le aportan un plus de versatilidad, pero “shields” oficiales solo existen los siguientes:

Arduino Ethernet Shield

Este shield está pensado para los que le quieren añadir a la placa Arduino UNO la capacidad de conectarse a una red cableada TCP/IP. Aporta la misma funcionalidad que la placa Arduino Ethernet pero en forma de shield complementario acoplado a la placa Arduino UNO. De hecho, este shield tiene el

mismo chip controlador W5100 que la placa Arduino Ethernet, y se configura con la misma librería de programación: la librería "Ethernet", la cual ya viene por defecto en el lenguaje Arduino.

Una vez conectado este shield sobre la placa UNO gracias a la ristra de pines que encaja perfectamente arriba y abajo, para nuestros circuitos utilizaremos a partir de entonces las entradas y salidas ofrecidas por los pines-hembra del shield Ethernet. Estas entradas y salidas tienen exactamente la misma disposición y funcionalidad que las de la placa UNO. Incluso si fuera necesario, se podría conectar sin problemas un segundo shield en la parte superior del shield Ethernet para seguir sumando funcionalidad.

El procedimiento de cargar nuestros programas en el microcontrolador de la placa UNO acoplada al shield Ethernet no varía respecto al realizado normalmente con una placa UNO independiente: primero debemos conectar la placa UNO a nuestro computador mediante el cable USB, y una vez cargado el programa, como siempre, podremos seguir alimentando la placa vía USB o bien desconectarla del computador y enchufarla a una fuente de alimentación externa. A partir de entonces, al conector RJ-45 del shield le podremos conectar un cable de red (técnicamente, un cable de par trenzado de categoría 5 o 6) de tipo "estándar" si deseamos comunicar el shield a un switch o un router o de tipo "cruzado" si deseamos comunicar el shield directamente a un computador.

Este shield requiere 5 V para funcionar. Este voltaje lo aporta la placa UNO mediante el encaje del pin de alimentación correspondiente entre placa y shield. La comunicación entre el chip W5100 y la placa UNO se establece mediante los pines 10,11,12 y 13 (vía SPI) por lo que estos pines no se pueden utilizar para otro propósito. Esto implica que realmente al usar este shield tenemos 4 entradas/salidas digitales menos.

Este shield también incorpora (tal como lo hace la placa Arduino Ethernet) un zócalo para colocar una tarjeta microSD, la cual se podrá utilizar mediante la librería de programación "SD", que viene por defecto en el lenguaje Arduino. Al igual que ocurre con el chip W5100, para poder comunicarse con la tarjeta microSD la placa Arduino UNO utiliza el protocolo SPI, pero para ello esta vez emplea como pin SS el número 4 en vez del 10 (reservado al W5100). Es decir, emplea los pines 4, 11, 12 y 13.

El hecho de compartir el canal SPI entre el chip W5100 y la tarjeta microSD implica que no se pueden utilizar estos dos dispositivos a la vez. Si en nuestros programas queremos utilizar ambos elementos, deberemos tener en cuenta esto para programar nuestro código de forma correcta. Si, en cambio, no queremos utilizar uno de los dos dispositivos, hay que tener la precaución de desactivarlo

explícitamente en nuestro código; para desactivar la tarjeta microSD se ha de configurar el pin 4 como salida y escribir un valor HIGH –ALTO– y para desactivar el chip W5100 se ha de hacer lo mismo pero con el pin 10.

Al igual que la placa Arduino Ethernet, este shield también tiene la posibilidad de acoplársele un módulo PoE.

Finalmente, indicar que este shield también dispone de su propio botón de "reset", el cual reinicia tanto el chip W5100 como la propia placa Arduino, y de una serie de LEDs informativos interesantes de conocer: "PWR" (indica que la placa y el shield reciben alimentación eléctrica), "LINK" (indica la presencia de una conexión de red, y parpadea cuando el shield recibe o transmite datos), "FULLD" (indica que la conexión de red es "full duplex"), "100M" (indica la presencia de una conexión de red de 100 megabits/s –en vez de una de 10 megabits/s–), "RX" (parpadea cuando el shield recibe datos),"TX" (parpadea cuando el shield envía datos) y "COLL" (parpadea cuando se detectan colisiones de paquetes en la red).

Arduino Wireless SD Shield

Este shield está pensado para permitir a una placa Arduino UNO poderse comunicar inalámbricamente mediante el uso de un módulo XBee (adquirido aparte) o similar. Esto permite establecer un enlace con otro dispositivo XBee a una distancia de hasta unos 100 metros en interior y de hasta unos 300 metros en exterior con línea de visión directa.

Igual que ocurre con el resto de shields oficiales, una vez conectado este shield sobre la placa UNO gracias a la ristra de pines que encaja perfectamente arriba y abajo, para nuestros circuitos utilizaremos a partir de entonces las entradas y salidas ofrecidas por los pines-hembra de este shield. Estas entradas y salidas tienen exactamente la misma disposición y funcionalidad que las de la placa UNO e, incluso si fuera necesario, se podría conectar sin problemas un segundo shield en la parte superior de este shield.

Este shield dispone de un conmutador etiquetado como "Serial Select" que sirve para determinar cómo se comunicará el módulo XBee con la placa Arduino sobre la que se encuentra. Si el conmutador esté colocado en la posición "Micro", los datos recibidos por el módulo XBee llegarán sin intermediarios al microcontrolador (señal RX) y los datos enviados desde el microcontrolador (señal TX) serán transmitidos tanto hacia el módulo como hacia el posible computador que esté conectado vía USB. No obstante, en esa posición el ATmega328P no podrá ser programado a través de USB.

Si el conmutador está colocado en la posición “USB”, el módulo XBee se comunicará directamente con el chip Atmega16U2 (suponiendo que estamos usando la placa UNO) ignorando completamente la presencia del microcontrolador ATmega328P. En esta posición, el módulo XBee será capaz de comunicarse directamente con un computador conectado vía USB para poder ser configurado y utilizado con total independencia del resto de elementos de la placa Arduino. No obstante, para que el modo “USB” pueda funcionar correctamente, hay que tener la precaución de programar previamente el microcontrolador ATmega328P con un código Arduino con sus funciones “setup()” y “loop()” totalmente vacías (en el capítulo 4 se verá qué significa esto).

Este shield también incorpora un zócalo para colocar una tarjeta microSD, la cual se podrá utilizar mediante la librería de programación “SD”, que viene por defecto en el lenguaje Arduino. Para poder comunicarse con esta tarjeta, la placa Arduino UNO utiliza el protocolo SPI, empleando para ello como pin SS el número 4, además de los pines 11, 12 y 13. Esto quiere decir que estos cuatro pines del shield no se pueden utilizar para otra cosa y por lo tanto, disponemos de 4 entradas/salidas digitales menos.

Arduino Wireless Proto Shield

Este shield es exactamente igual al shield anterior, pero sin tener el zócalo microSD.

Arduino WiFi Shield

Este shield está pensado para los que le quieren añadir a la placa Arduino UNO la capacidad de conectarse inalámbricamente a una red TCP/IP. Incorpora el chip HDG104 del fabricante H&D Wireless, el cual incluye una antena integrada y permite conectarse a redes Wi-Fi de tipo 802.11b y de tipo 802.11g. También incorpora el chip ATmega 32UC3, un microcontrolador de 32 bits que en este shield viene preprogramado de fábrica para proporcionar una pila IP completa (TCP y UDP).

Las redes a las que se puede conectar pueden ser abiertas, o bien estar protegidas mediante encriptación de tipo WEP o WPA2-Personal (no puede conectarse a redes WPA2-Enterprise). En cualquier caso, solo podrá conectarse a una red si esta emite públicamente su SSID (es decir, si su SSID no está oculto). Para poder gestionar este shield, se debe utilizar la librería de programación oficial “WiFi”, la cual viene por defecto en el propio lenguaje Arduino.

Igual que ocurre con el resto de shields oficiales, una vez conectado este shield sobre la placa UNO gracias a la ristra de pines que encaja perfectamente arriba y abajo, para nuestros circuitos utilizaremos a partir de entonces las entradas y salidas ofrecidas por los pines-hembra de este shield. Estas entradas y salidas tienen exactamente la misma disposición y funcionalidad que las de la placa UNO. Incluso si fuera necesario, se podría conectar sin problemas un segundo shield en la parte superior de este shield para seguir sumando funcionalidad.

El procedimiento de cargar nuestros programas en el microcontrolador de la placa UNO acoplada al shield WiFi no varía respecto al realizado normalmente con una placa UNO independiente: simplemente deberemos conectar la placa UNO a nuestro computador mediante el cable USB para cargar el programa, y una vez hecho esto, como siempre, podremos seguir alimentando el conjunto placa más shield vía USB o bien desconectarla del computador y enchufarla a una fuente de alimentación externa. El shield requiere 5 V para funcionar, y este voltaje lo aporta la placa UNO mediante el encaje del pin de alimentación correspondiente entre placa y shield.

Este shield incorpora (tal como lo hace el shield Ethernet o el Wireless SD) un zócalo pensado para colocar una tarjeta microSD. La idea es que esta tarjeta sirva para almacenar ficheros y hacerlos disponibles a través de la red, mediante el uso de la librería de programación “SD” (que viene por defecto en el lenguaje Arduino). La comunicación entre el chip HDG104 y la placa UNO se establece mediante los pines 10, 11, 12 y 13 (vía SPI) por lo que estos pines no se pueden utilizar para otro propósito. La comunicación entre la tarjeta microSD y la placa Arduino UNO también se establece mediante el protocolo SPI, pero para ello esta vez emplea como pin SS el número 4 (en vez del 10 usado por el HDG104). Además de todo ello, el pin número 7 también está reservado para la comunicación interna entre la placa y el shield WiFi. Por tanto, hay que tener en cuenta que disponemos de 6 pines menos (el 4, 7, 10, 11, 12 y 13) ya que no podremos utilizarlos como entradas o salidas estándares.

Además, el hecho de compartir el canal SPI entre el chip HDG104 y la tarjeta microSD implica que no se pueden utilizar estos dos dispositivos a la vez. Si en nuestros programas queremos utilizar ambos elementos, deberemos tener en cuenta esto para programar nuestro código de forma correcta. Si, en cambio, no queremos utilizar uno de los dos dispositivos, hay que tener la precaución de desactivarlo explícitamente en nuestro código; para desactivar la tarjeta microSD se ha de configurar el pin 4 como salida y escribir un valor HIGH –ALTO– y para desactivar el chip HDG104 se ha de hacer lo mismo pero con el pin 10.

Por otro lado, indicar que este shield también dispone de su propio botón de “reset”, el cual reinicia tanto el chip HDG104 como la propia placa Arduino, y también de una serie de LEDs informativos interesantes de conocer: el “L9” (conectado directamente al pin digital nº 9), el “LINK” (que indica que se ha establecido conexión a una red), el “ERROR” (que indica si ha habido un error en la comunicación) y el “DATA” (que indica que hay datos transmitiéndose/recibiéndose en ese momento).

Finalmente, podemos observar que este shield incluye un conector USB de tipo mini-B. Este conector no es para programar la placa Arduino, sino para realizar una tarea algo más avanzada que en general no será necesaria y que por tanto, no estudiaremos en este libro: actualizar (mediante el protocolo DFU) el firmware del chip ATmega32U3. Una utilidad de esto es el poder añadir de forma directa al 32U3 protocolos de red más complejos sin depender del limitado espacio que proporciona el ATmega328 de la placa Arduino. De hecho, incluso se podría alterar el firmware de tal forma que el shield WiFi podría funcionar como un dispositivo independiente sin necesidad de tener una placa Arduino conectada a él.

Este shield también dispone de un zócalo FTDI (al cual se le puede conectar un adaptador o cable USB-Serie) para poder interactuar directamente con el chip 32U3 y obtener información de diagnóstico. Para ello, se debe utilizar un programa de computador especial que permita el envío de comandos al shield para que este los interprete y ejecute. Es decir, un programa que funcione como “terminal serie”. Este tipo de programas los estudiaremos en el próximo capítulo. La lista de comandos la podemos consultar aquí: http://arduino.cc/en/Hacking/WiFiShield32USerial .

Arduino Motor Shield

Este shield incorpora el chip L298P del fabricante STMicroelectronics; la “P” final simplemente indica el tipo de encapsulado que tiene, ya que para el mismo chip L298 existen otras formas y tamaños, identificados con otras letras). Este chip está diseñado para controlar componentes que contienen inductores –”bobinas”– en su estructura interna, tales como relés, solenoides, motores de corriente continua –DC– o motores paso a paso – “steppers” –, entre otros.

En concreto, gracias al chip L298P, el shield Arduino Motor nos permite dominar la velocidad y sentido de giro de hasta dos motores DC de forma independiente o bien estas dos magnitudes de un motor paso a paso. También podremos realizar diferentes medidas sobre las capacidades de los motores conectados. Para saber más sobre los distintos tipos de motores, sus propiedades y aplicaciones, recomiendo consultar el apartado dedicado a ellos en el capítulo 5.

Como el voltaje necesario para el correcto funcionamiento de los motores suele sobrepasar el aportado por un simple cable USB, este shield siempre necesitará ser alimentado por una fuente de externa. Esta puede ser bien un adaptador AC/DC que aporte entre 7 y 12 V DC de salida y conectado al jack de 2,1 mm de la placa Arduino UNO, o bien una pila (preferiblemente de 9 V) conectada mediante cables directamente a los bornes de los tornillos etiquetados como "Vin" y "Gnd" del propio shield. En ambos casos (usando el adaptador AC/DC o la pila), estaríamos alimentando a la vez tanto al shield como a la placa.

No obstante, si los motores utilizados requieren más de 9 V, necesitaremos separar las líneas de alimentación de placa y shield de manera que cada una se alimente por separado: esto se hace cortando la soldadura existente entre los extremos del jumper etiquetado como "Vin connect" que aparece en el dorso del shield. De esta forma, el shield se podría continuar alimentando mediante la fuente conectada a sus bornes de tornillo, y la placa por su lado se alimentaría o bien con el cable USB o bien con la fuente externa que estuviera conectada al jack de 2,1 mm. De todas formas, hay que tener en cuenta que el máximo voltaje que admiten los bornes de tornillo es de 18 V.

Este shield tiene dos canales separados, etiquetados como "A" y "B", en forma de bornes de tornillo para conectar allí los motores. Cada canal por separado puede manejar un motor DC independiente, pero también se pueden combinar para manejar entre los dos un único motor paso a paso. Si estamos en el primer caso (es decir, si queremos manejar uno o dos motores DC), podemos emplear determinados pines-hembra de entrada/salida para controlar y monitorizar el motor DC conectado a cada canal. Esto implica que esos pines-hembra no los podremos usar para otra cosa, por lo que dispondremos de menos entradas/salidas de propósito general. En concreto, la función estos pines-hembra especializados es la siguiente:

Función	**Pines Canal A**	**Pines Canal B**
Sentido de movimiento	D12	D13
Velocidad (PWM)	D3	D11
Freno	D9	D8
Detección de corriente	A0	A1

A partir de la tabla anterior se puede deducir cómo es el funcionamiento del shield. Después de conectar el par de cables que proporcione cada motor a los terminales (+) y (-) de sus respectivos bornes de tornillo (canal A o canal B), para controlar su sentido de movimiento se deberá enviar una señal de valor HIGH o LOW

(según el sentido de giro deseado) a los pines de dirección correspondientes. Para controlar la velocidad de giro se deberá variar los valores de la señal PWM en los pines correspondientes. Finalmente, los pines de freno si tienen el valor HIGH paran en seco los motores DC asociados (en vez de dejarlos desacelerar paulatinamente como ocurriría si se les corta la alimentación).

Se puede conocer el valor de la intensidad de corriente que pasa a través del motor DC leyendo el valor de la señal existente en los pines correspondientes de la tabla anterior, ya que la señal recibida es proporcional a dicha intensidad. Cada canal puede recibir una corriente de 2 amperios como máximo, la cual se corresponde a una señal máxima posible de 3,3 V.

Si no necesitamos la funcionalidad de freno o de detección de corriente y necesitamos más pines para nuestro proyecto, se pueden desactivar estas funcionalidades cortando la soldadura existente entre los extremos de los respectivos jumpers etiquetados convenientemente en el dorso del shield.

Este shield dispone además de una serie de conectores muy interesantes: 2 conectores blancos de tipo “TinkerKit” de 3 pines para poder enchufar dos entradas analógicas (conectadas internamente a los pines A2 y A3); 2 conectores naranja de tipo “TinkerKit” de 3 pines para poder enchufar dos salidas analógicas (conectadas internamente a las salidas PWM de los pines D5 y D6) y 2 conectores blancos de tipo “TinkerKit” de 4 pines, uno para entrada I^2C/TWI y otro para salida I^2C/TWI.

Tinkerkit (y otros)

“TinkerKit” (http://www.tinkerkit.com) es un conjunto de sensores y actuadores prefabricados que comparten una forma coherente y común de acoplarse a cualquier circuito compatible, de manera que la creación de proyectos sea tan sencilla como conectar y desconectar módulos TinkerKit igual que si fueran piezas de un puzle. De esta manera, se pueden poner en marcha rápidamente entornos interactivos sin tener que soldar ni usar ni siquiera una breadboard. Todos los componentes TinkerKit están montados sobre un soporte generalmente naranja y todos disponen sin distinción del mismo cable de 3 (o 4) pines, el cual permite transmisiones de datos a distancias de hasta 5 metros. Entre los componentes TinkerKit podemos encontrar botones, LEDs de diversos colores, potenciómetros rotatorios y lineales, joysticks, acelerómetros, sensores de movimiento, de luz, de infrarrojos, de temperatura, transistores, servomotores, brújulas, GPS, sistemas táctiles...

Para que nuestra placa Arduino UNO pueda reconocer y trabajar con componentes TinkerKit, en general lo más recomendable es utilizar el llamado "Sensor Shield", un shield preparado para poderle enchufar allí de una forma centralizada los componentes TinkerKit que deseemos. No obstante, una vez conectados los módulos TinkerKit al "Sensor shield" y este a una placa Arduino, para poder gestionar todo el sistema es necesario utilizar además una librería de programación especial llamada "TinkerKit", descargable de la página de TinkerKit (ya que no viene por defecto incorporada en el lenguaje Arduino). Mediante esta librería podemos manipular dentro de nuestro código cada uno de los módulos TinkerKit de forma independiente, gracias a una serie de instrucciones particulares que facilitan su control.

Existen otros sistemas de componentes que persiguen la misma filosofía de "enchufar y listo", pero cada uno adopta sus propios conectores y pueden no ser compatibles entre sí. Entre las alternativas a Tinkerkit podemos encontrar por ejemplo el "GROVE Starter Kit" (producto con código ELB152D2P) del distribuidor Seeedstudio. Este producto incorpora varios módulos GROVE, entre los cuales podemos encontrar sensores de luz, de temperatura, de agua, de gas, de movimiento, de distancia, de sonido, de electricidad, sensores táctiles, acelerómetros, brújulas, pantallas de cristal líquido, relojes de tipo RTC, receptores/transmisores Bluetooth, GPS… además de potenciómetros, joysticks, botones, LEDs, zumbadores, altavoces, etc. Incluye además el llamado "Grove Base Shield", que no es más que un shield acoplable a una placa Arduino con conectores de tipo GROVE donde se pueden enchufar diferentes módulos de este tipo. Para controlar los módulos GROVE, al contrario que con TinkerKit, no es necesario utilizar ninguna librería específica: con el lenguaje Arduino tal cual ya es suficiente.

Seeedstudio también comercializa otro kit de módulos (con la misma filosofía pero tal vez no tan completo) llamado "Electronic Bricks Kit" (producto con código ELB138E1P), el cual incluye el shield "Arduino Sensor Shield". Otro "Sensor Shield" independiente pero compatible con los "Electronic Bricks" es el distribuido por Yourduino.

Arduino Proto Shield

Este shield permite de forma fácil diseñar e implementar nuestros circuitos personales. Básicamente lo que hace es ofrecer un área de trabajo donde se pueden soldar (usando tanto la técnica del montaje superficial –SMT–, como la técnica de los agujeros pasantes –en inglés THT, de "through-hole technology"–) los diferentes componentes electrónicos que necesitemos para montar nuestro proyecto. De esta

forma, podemos tener de una forma muy compacta todo un circuito completo formado por placa, shield y componentes conectados, todo en uno.

La tecnología THT es tal vez la manera más sencilla de soldar un componente a una placa de circuito impreso. Para ello, se requiere que los componentes dispongan de patillas de metal de varios milímetros de longitud y la PCB disponga de orificios taladrados: la técnica simplemente consiste en atravesar con cada patilla un orificio de la placa y seguidamente soldar cada una de estas patillas a la cara inferior de la PCB (cortando opcionalmente, una vez realizada la soldadura, los milímetros de patilla que puedan sobresalir por debajo). En cambio, con la tecnología SMT los componentes no poseen patillas sino pequeñas pestañas metálicas que son soldadas sobre unos pequeños zócalos especiales presentes sobre la capa exterior de las placas. La técnica SMT, comparándola con la THT, permite que los componentes puedan ser mucho más pequeños, más baratos y que se puedan utilizar en ambos lados de las placas (permitiendo así una densidad de componentes mucho mayor), pero en general es más difícil de soldar para los electrónicos aficionados.

Aclaremos aquí ya que en este libro no se realizará ningún proyecto donde sea necesario soldar componentes. Si bien es cierto realizar soldaduras domésticas (sobre todo de tipo THT) no es excesivamente complicado y pueden conseguirse resultados muy buenos con material y herramientas relativamente baratas sin problemas, hay que reconocer que se requiere cierta experiencia y pericia en este tema. Por eso, y ya que este texto no deja de ser una introducción al mundo de la electrónica, no se ha considerado oportuno profundizar en este asunto, que tal vez pertenezca a un peldaño inmediatamente superior de conocimientos.

Aun si, no se desea soldar ningún componente (como es nuestro caso), este shield también nos puede ser útil porque permite colocar permanentemente en un área especialmente reservada para ello una pequeña breadboard de 1,8 x 1,4 pulgadas (adquirida aparte). Así podremos enchufar rápidamente todos los elementos del circuito y comprobar que este funcione correctamente de una forma muy compacta y segura.

Este shield recibe la alimentación de los pines estándares 5 V y GND de la placa Arduino y la reparte a lo largo de dos filas de once agujeros etiquetadas en el shield claramente como “5 V” y “GND” . A partir de estas dos filas se puede alimentar fácilmente cualquier componente (incluso zócalos DIP) que se suelde a la placa.

Otras características de este shield son que dispone de un botón de reinicio y de un conector ICSP (comunicado mediante los pines 10, 11, 12 y 13 a la placa

Arduino UNO) para que no tengamos que usar los ubicados en la placa base, y también que replica la disposición exacta de todos los pines de E/S de la placa Arduino UNO para poder trabajar con ellos de la forma habitual.

¿QUÉ SHIELDS NO OFICIALES EXISTEN?

Existe gran cantidad de shields diseñados y construidos por la comunidad que ofrecen soluciones a necesidades específicas que los shields oficiales no presentan. En este sentido, es muy recomendable consultar la web http://www.shieldlist.org, que ofrece una lista centralizada de prácticamente todos los modelos de shields existentes, clasificados por fabricante. Allí encontraremos además las especificaciones básicas de cada shield, los detalles de sus conexiones y en general toda la información necesaria para comprender su funcionamiento.

Un ejemplo de shield que nos puede ser útil en varios proyectos diferentes es el llamado genéricamente “proto shields”. Básicamente, lo que ofrecen este tipo de shields es un área de prototipado para implementar los circuitos de una forma mucho más compacta que si lo hiciéramos mediante una breadboard por separado (además de algún pulsador, LED o potenciómetros extra, según el modelo). Un ejemplo de este tipo de shield lo acabamos de ver: el Arduino Proto Shield oficial, pero otros similares fabricados por terceros son, por ejemplo, el “Protoshield Pro” de Freetronics o el “Prototyping Shield for Arduino” de DFRobot.

Desgraciadamente, muchos “proto shields” se distribuyen en forma de kit (es decir, con sus piezas sin ensamblar), por lo que es necesario, una vez adquiridos, soldar todas sus partes para obtener un shield funcional. Ejemplos de estos últimos son el “MakerShield Kit” de Makershed, el “Proto Shield for Arduino Kit” de Adafruit, el “Protoshield Kit for Arduino” de Seeedstudio o el “Arduino Protoshield Kit” de Sparkfun.

Una alternativa a los “proto shields” anteriores son los llamados genéricamente “screw shields”, los cuales sustituyen los pines-hembra por terminales de tuerca de 3,5 mm, ideales para fijar los cables de una forma mucho más estable mediante destornilladores. Ejemplos son el “Terminal Shield for Arduino” de Freetronics, el “ProtoScrewShield” de Sparkfun (en forma kit), el “Proto-ScrewShield” de Adafruit (en forma de kit), el “Arduino Proto Screw Shield” de IteadStudio o el “Power ScrewShield” de Snootlab.

Otro tipo de shields diferentes de los anteriores pero que conviene conocer porque también nos pueden ser de gran ayuda en múltiples proyectos son los que permiten situar sobre sí mismos algún tipo de batería de tal forma que se consiga una estructura sólida y compacta de fuente de alimentación más placa Arduino, haciendo al conjunto energéticamente independiente de nuestro computador. Un ejemplo de este tipo de shield es el “Lithium BackPack” de LiquidWare, el cual admite la colocación (mediante conector JST) de una batería Li-Ion de varios tipos compatibles (2200 mAh, 1000 mAh o 660 mAh), todos ellos capaces de alimentar nuestra placa Arduino; también tiene la opción de actuar como cargador de la batería, bien a través de la alimentación recibida por la placa Arduino o bien directamente del exterior mediante un conector USB mini-B ya incluido.

Otros shield similar al anterior es el “LiPower Shield” de Sparkfun. Este shield también es capaz también de realizar dos funciones: se puede utilizar como cargador de una batería (en este caso, de tipo LiPo de 3,7 V y preferiblemente 500 mAh) y además como fuente de alimentación de la placa Arduino. La circuitería interna del shield se encarga de proteger la batería de cargas, descargas o corrientes excesivas, y de convertir los 3,7 V ofrecidos por ella en los 5 V necesarios para hacer funcionar Arduino, por lo que tan solo nos deberemos preocupar de conectar la batería LiPo al zócalo de tipo JST incorporado en el shield y ya está. Si se desea recargar la batería, lo podemos hacer a través de la alimentación recibida por la propia placa Arduino o bien directamente del exterior mediante un conector USB mini-B que viene incluido. Este shield permite además monitorizar el estado de carga de la batería mediante un código Arduino específico, descargable de la página de este producto.

De todas formas, es imposible nombrar aquí todos los shields que puedan ser destacables por alguna razón: son simplemente demasiados y muy diferentes entre sí. Muchos de ellos los iremos conociendo a medida que necesitemos utilizarlos en los diferentes proyectos existentes a lo largo del libro, pero aun así quedarán en el tintero muchos que nos puede ser útil conocer. Recomiendo, pues, visitar la web mencionada anteriormente y/o consultar el extenso catálogo de los fabricantes y diseñadores de shields más importantes, como Sparkfun, Adafruit, Seeedstudio, Iteadstudio, Freetronics, Olimex o DFRobot, entre otros listados en el apéndice A.

3 SOFTWARE ARDUINO

¿QUÉ ES UN IDE?

Un programa es un conjunto concreto de instrucciones, ordenadas y agrupadas de forma adecuada y sin ambigüedades que pretende obtener un resultado determinado. Cuando decimos que un microcontrolador es “programable”, estamos diciendo que permite grabar en su memoria de forma permanente (hasta que regrabemos de nuevo si es necesario) el programa que deseemos que dicho microcontrolador ejecute. Si no introducimos ningún programa en la memoria del microcontrolador, este no sabrá qué hacer.

Las siglas IDE vienen de Integrated Development Environment, lo que traducido a nuestro idioma significa Entorno de Desarrollo Integrado. Esto es simplemente una forma de llamar al conjunto de herramientas software que permite a los programadores poder desarrollar (es decir, básicamente escribir y probar) sus propios programas con comodidad. En el caso de Arduino, necesitamos un IDE que nos permita escribir y editar nuestro programa (también llamado “sketch” en el mundo de Arduino), que nos permita comprobar que no hayamos cometido ningún error y que además nos permita, cuando ya estemos seguros de que el sketch es correcto, grabarlo en la memoria del microcontrolador de la placa Arduino para que este se convierta a partir de entonces en el ejecutor autónomo de dicho programa.

Para poder empezar a desarrollar nuestros propios sketches (o probar alguno que tengamos a mano) deberemos instalar en nuestro computador el IDE que nos proporciona el proyecto Arduino. Para ello, seguiremos alguno de los pasos mostrados a continuación, según cada caso particular.

INSTALACIÓN DEL IDE ARDUINO

Ubuntu

La manera más fácil de instalar el IDE de Arduino en Ubuntu es mediante el uso de su "Centro de Software" (o cualquier otro gestor de paquetes equivalente, tal como Synaptic o apt-get/aptitude). El paquete que necesitamos se llama "arduino" y estará disponible si tenemos activado el repositorio "universe". Si queremos usar el terminal de comandos para instalar el IDE de Arduino, podríamos escribir por tanto:

```
sudo apt-get -y install arduino
```

Es interesante notar que Ubuntu divide el entorno de programación Arduino en dos paquetes diferentes: el llamado "arduino", que incluye los ficheros que conforman la interfaz gráfica del IDE y el llamado "arduino-core", que incluye las herramientas fundamentales de compilación, programación y grabación de sketches. Estas últimas son las que realizan todo el trabajo importante y son invocadas mediante la interfaz gráfica del IDE (el cual en este sentido hace de mero intermediario entre ellas y el usuario), aunque también pueden ser ejecutadas directamente mediante el intérprete de comandos. Instalando el paquete "arduino", se instalará de forma automática el paquete "arduino-core" sin necesidad de especificarlo (es de hecho lo que hacemos con el comando anterior), pero a la inversa no es así: si quisiéramos trabajar mediante un terminal sin usar el IDE propiamente dicho podríamos instalar el paquete "arduino-core" autónomamente sin necesidad de instalar el paquete "arduino".

Fedora

La manera más fácil de instalar el IDE de Arduino en Fedora es mediante el uso de cualquiera de sus gestores de paquetes disponibles (PackageKit, Yum, etc). El paquete que necesitamos se llama "arduino" y está disponible en el repositorio oficial de Fedora. Si usamos el terminal de comandos, para instalar el IDE de Arduino debemos escribir: `sudo yum -y install arduino`.

Al igual que ocurría con Ubuntu, Fedora divide el entorno de programación Arduino en dos paquetes: “arduino” y “arduino-core”. Además, ofrece un tercer paquete llamado “arduino-doc” (instalado automáticamente mediante el comando anterior), que contiene toda la documentación y ejemplos de programación oficial de Arduino. En Ubuntu esta información se encuentra incluida dentro del paquete “arduino-core”.

Cualquier sistema Linux

Para instalar el IDE de Arduino en una distribución cualquiera de Linux, debemos ir a http://arduino.cc/en/Main/Software, la página de descargas oficial de Arduino. Allí, bajo el apartado “Downloads”, aparecen dos enlaces diferentes para descargar la versión del IDE Arduino para Linux. Ambos enlaces apuntan a un archivo comprimido en formato “tgz” alojado en el almacén de software oficial de Arduino (ubicado en http://code.google.com/p/arduino), pero un enlace corresponde a la versión del IDE de 32 bits y el otro a la de 64 bits. Es necesario, por tanto, conocer primero qué tipo de sistema operativo tenemos instalado (32 bits o 64 bits) y descargarnos entonces el paquete “tgz” correspondiente.

Para saber de manera rápida si nuestro sistema Linux es de 32 bits o de 64 bits podemos abrir un terminal y escribir el siguiente comando: `getconf LONG_BIT` La salida que obtengamos como resultado de ejecutar el comando anterior será “32” si nuestro sistema es de 32 bits y “64” si nuestro sistema es de 64 bits.

Una vez sabido esto, ya podemos elegir el paquete “tgz” adecuado y, una vez descargado, lo descomprimiremos. Cuando entremos dentro de la carpeta descomprimida que obtengamos, no encontraremos ningún instalador ni nada parecido (solo una estructura de archivos y subcarpetas que no deberemos modificar para nada) porque de hecho, nuestro IDE ya está listo para ser usado desde este mismo momento: tan solo deberemos clicar sobre el archivo ejecutable (en realidad, un shell script con permisos de ejecución) llamado “arduino” y, después de confirmar la opción “Ejecutar” que aparecerá en el cuadro emergente, veremos el IDE finalmente ante nosotros.

También podemos ejecutar el IDE a través del intérprete de comandos; para ello deberemos situarnos mediante el comando `cd` dentro de la carpeta donde se encuentra el shell script y escribir `./arduino` (importante no olvidarse del punto y la barra).

Evidentemente, podremos mover y guardar en el lugar que deseemos de nuestro disco duro la carpeta contenedora de los ficheros que forman el IDE, siempre y cuando no alteremos nada de su interior. Seguramente también deseemos crear un acceso directo del ejecutable allí donde lo consideremos oportuno (por ejemplo, en el escritorio).

En realidad todo este proceso que acabamos de describir en este apartado también se podría haber seguido igualmente si nuestra distribución Linux fuera Ubuntu o Fedora. De hecho, puede haber ocasiones donde nos interese más seguir este procedimiento "manual" que utilizar los repositorios respectivos: una de estas ocasiones podría ser la aparición en la web oficial de una versión más actualizada del IDE que todavía no haya sido empaquetada ni introducida en los repositorios de nuestra distribución preferida.

Las dependencias

Desgraciadamente, es posible que al intentar ejecutar el IDE tal como se describe en este apartado, obtengamos diferentes errores debidos a no tener previamente instalados en el sistema los paquetes necesarios para poder ejecutar el script "arduino" sin problemas (es decir, por no tener instaladas las "dependencias" del IDE Arduino). Si usamos un gestor de paquetes cualquiera para instalar el IDE desde los repositorios de nuestra distribución (como el "apt-get" de Ubuntu o el "yum" de Fedora) no tendremos este inconveniente porque precisamente una de las tareas de un gestor de paquetes es detectar si las dependencias necesarias están ya instaladas (y si no lo están, proceder a su instalación automática).

Para resolver los posibles errores de falta de dependencias, deberemos instalarlas de alguna manera antes de volver a intentar ejecutar el IDE Arduino. Lo más sencillo es hacerlo a través de nuestro gestor de paquetes favorito, pero ¿qué dependencias son? A continuación, mostramos como referencia una lista (¡no exhaustiva!) con los nombres de los paquetes para Ubuntu y Fedora de las dependencias básicas del IDE Arduino, además de una breve explicación de su utilidad:

PAQUETE UBUNTU: openjdk-7-jre
PAQUETE FEDORA: java-1.7.0-openjdk
UTILIDAD: conjunto de utilidades y librerías que permiten la ejecución de aplicaciones escritas en el lenguaje de programación Java. Precisamente el código del IDE (a diferencia del de las herramientas básicas de línea de comandos), está desarrollado con este lenguaje, por lo que la instalación de este paquete es imprescindible para su correcto funcionamiento en cualquier sistema operativo. Su web oficial es: http://openjdk.java.net

PAQUETE UBUNTU: librxtx-java
PAQUETE FEDORA: rxtx
UTILIDAD: **l**ibrería escrita en Java que permite comunicar aplicaciones escritas en ese mismo lenguaje (en nuestro caso, el IDE Arduino) con dispositivos externos mediante comunicación serie (en nuestro caso, las placas Arduino). En pocas palabras, es la pieza de software que permite al IDE poder comunicarse a través del cable USB con la placa que tengamos conectada. Su web oficial es: http://rxtx.qbang.org.

PAQUETE UBUNTU: gcc-avr
PAQUETE FEDORA: avr-gcc y avr-gcc-c++
UTILIDAD: **c**olección de utilidades que transforman un código fuente escrito en el lenguaje de programación C (o C++) en un fichero con el formato binario adecuado (concretamente el formato ".hex") para poder ser grabado cuando se desee en la memoria de un microcontrolador AVR de Atmel. Es decir, en un conjunto de programas que transforman nuestro sketch en un fichero entendible y ejecutable por este tipo de microcontroladores. Hablando técnicamente, "gcc-avr" es un compilador cruzado (es decir, un compilador que genera programas ejecutables para otra plataforma diferente –Arduino– de la plataforma donde está instalado –PC–). Sobre el concepto de compilador hablaremos más adelante. Su web oficial es http://gcc.gnu.org .

El paquete para Ubuntu incluye los compiladores cruzados para AVR del lenguaje C y del lenguaje C++ pero en Fedora se separan en dos paquetes diferentes. Posiblemente sorprenda hablar de los lenguajes C y C++, pero ya se ha comentado anteriormente que el lenguaje Arduino no deja de ser un subconjunto más sencillo basado en estos dos lenguajes (muy parecidos entre sí, de hecho), por lo que en realidad, cuando escribimos nuestros sketches en lenguaje Arduino, lo estamos haciendo sin saberlo básicamente en un lenguaje C/C++ simplificado. Por ello, las herramientas básicas de compilación son las propias de estos dos lenguajes.

PAQUETE UBUNTU: avr-libc
PAQUETE FEDORA: avr-libc
UTILIDAD: conjunto de librerías necesarias para que "gcc-avr" pueda realizar las compilaciones correctamente. Técnicamente, son las librerías estándar del lenguaje C para la plataforma AVR. Su web oficial es http://www.nongnu.org/avr-libc

Como complemento de “gcc-avr” (que se instala automáticamente como dependencia) también debemos tener en el sistema el conjunto de utilidades llamadas genéricamente “binutils” (el paquete correspondiente en Ubuntu se llama “binutils-avr” y en Fedora “avr-binutils”), las cuales realizan tareas de ensamblaje y enlace entre las librerías “avr-libc” y otras librerías necesarias para lograr una correcta compilación. Su web oficial es: http://www.gnu.org/software/binutils

PAQUETE UBUNTU: avrdude
PAQUETE FEDORA: avrdude
UTILIDAD: utilidad de línea de comandos que sirve para cargar desde un PC los ficheros compilados por “gcc-avr” (con la ayuda de “avr-libc” y “avr-binutils”) en la memoria de los microcontroladores de tipo AVR (con la ayuda del “bootloader” que tenga incorporado, o bien directamente a través de un programador ISP). Su web oficial es: http://www.nongnu.org/avrdude. En modelos de placas Arduino anteriores a las actuales en vez de utilizarse Avrdude venía como dependencia otra utilidad parecida llamada Uisp (http://www.nongnu.org/uisp). En el modelo Arduino Due se usa otra utilidad diferente: Bossa (http://www.shumatech.com/web/products/bossa).

Hay que aclarar que de todas las dependencias anteriores, existen unas cuantas que en realidad no deberemos descargar nunca “a mano” porque ya vienen incluidas dentro del paquete Arduino oficial (pero se han nombrado por completitud). La razón de esta inclusión es porque las versiones de los programas incorporados dentro del IDE son ligeramente diferentes de las versiones “estándar” disponibles en los repositorios, y de hecho, no se recomienda utilizar estos últimos porque el IDE no funcionará. Concretamente, son el paquete “rxtx” (modificado para soportar las placas que utilizan el chip ATmega16U2 en vez del chip FTDI, como la Arduino UNO), el paquete “avrdude” (modificado para asegurar un comportamiento correcto del auto-reinicio de las placas antes de la carga del programa, sin el cual dicha carga fallaría) y el paquete “avr-gcc”. El código fuente de todas estas versiones modificadas se encuentra disponible en http://github.com/arduino/Arduino

Los permisos de usuario

Además de instalar todas esas dependencias, aún deberemos realizar un último paso antes de poder ejecutar el IDE Arduino: asegurarnos de que nuestro usuario del sistema tenga los permisos necesarios para que al utilizar el IDE, este pueda comunicarse a través del cable USB con la placa. Si no hacemos esto, solamente ejecutando el IDE como usuario “root” podremos cargar programas en el microcontrolador, lo cual no es muy recomendable.

Para lograrlo, en Ubuntu simplemente tendremos que tener la precaución de añadir nuestro usuario al grupo de usuarios predefinido "dialout". Este grupo de usuarios predefinidos tiene ya establecidos los permisos correctos para poder establecer una comunicación serie con las placas Arduino que se conecten al computador. Si nuestro usuario no pertenece a este grupo (por defecto sí suelen pertenecer), se puede solucionar abriendo un intérprete de comandos y ejecutando: `sudo usermod -a -G dialout miusuario` , donde "miusuario" lo deberemos cambiar por el nombre de nuestro usuario del sistema, donde el parámetro "-G dialout" especifica el grupo al que lo queremos añadir y donde el parámetro -a indica que tan solo queremos añadir un nuevo grupo al usuario sin eliminarlo de los otros posibles grupos a los que pudiera pertenecer anteriormente.

En el caso de Fedora, deberemos comprobar que nuestro usuario pertenezca igualmente al grupo "dialout" y además, a otro grupo predefinido, el grupo "lock". Una vez asignados los grupos pertinentes a nuestro usuario, ya sea en Ubuntu o Fedora, deberemos reiniciar la sesión para que los cambios tengan efecto.

Sobre el reconocimiento y uso de dispositivos USB-ACM en Linux

Cuando se conecta una placa Arduino UNO a un computador que ejecuta un sistema Linux, ocurren varias cosas:

1. La placa (más concretamente el chip ATmega16U2) es reconocida como un dispositivo de tipo "USB ACM". Esto es fácil comprobarlo ejecutando el comando `dmesg` y observando su salida.

Independientemente del sistema operativo utilizado, los dispositivos USB de tipo ACM ("Abstract Control Modem") tienen la característica de poder establecer con el computador una comunicación serie sencilla y directa a través de USB. En realidad son un subconjunto de un conjunto mayor de dispositivos, los pertenecientes a la clase CDC ("Communications device class"), los cuales a su vez son un pequeño conjunto de todos los posibles chips que cumplen el estándar USB.

2. Se genera automáticamente el fichero /dev/ttyACM#, donde la "#" será un número diferente dependiendo de la cantidad de dispositivos del mismo tipo que haya conectados en ese momento (supondremos a partir de ahora que la placa se reconoce como /dev/ttyACM0). Igualmente, cuando la placa se desconecta del computador, el fichero /dev/ttyACM0 desaparece automáticamente. Es fácil comprobar la existencia de este fichero observando la salida de `dmesg` o también ejecutando el comando `ls -l /dev/ttyACM0`.

Para poder reconocer los periféricos de tipo “USB ACM” y generar el fichero de dispositivo /dev/ttyACM0 pertinente, el kernel Linux utiliza un módulo (que en las distribuciones más extendidas siempre viene integrado por defecto) llamado “cdc_acm”. Gracias a que este módulo está funcionando, podemos establecer una comunicación serie entre nuestro sistema Linux y el dispositivo representado por /dev/ttyACM0.

Si en una distribución Ubuntu o Fedora se observa la salida del comando “ls” especificado en el punto 2, se puede comprobar que el propietario de este archivo es el usuario “root” y el grupo propietario de este archivo es el grupo “dialout”, los cuales tienen permisos de lectura y escritura sobre el archivo. Esta es la razón por la que necesitamos incluir nuestro usuario del sistema en el grupo “dialout”: para poder tener permisos de lectura y escritura en ese fichero y así poder recibir y enviar datos a través de su dispositivo asociado. También podríamos asignar los permisos de lectura y escritura al grupo “otros” (con el comando “chmod”, por ejemplo) pero no es una solución demasiado elegante.

La razón de que Fedora además necesite que nuestro usuario pertenezca al grupo “lock” es porque en esta distribución, este grupo es el propietario y tiene todos los permisos del directorio /run/lock/lockdev, directorio utilizado por el kernel para bloquear un posible acceso simultáneo a la placa por varios usuarios del sistema. En definitiva: para lograr que nuestro usuario pueda comunicarse con ella, necesita tener permisos totales (de lectura, escritura y acceso –ejecución–) sobre ese directorio. En Ubuntu este paso no es necesario porque el directorio en cuestión ya tiene por defecto permisos totales para cualquier usuario del sistema (es por tanto una configuración de fábrica más permisiva).

No está de más saber que cualquier chip que cumpla con el estándar USB necesita tener un identificador de producto (“product id”, PID) y un identificador de fabricante (“vendor id”, VID) oficiales, propios y únicos para poder ser comercializado. Los códigos VID son vendidos a los diferentes fabricantes de hardware del mundo por la corporación internacional USB-IF (http://www.usb.org), encargada de la estandarización y especificación del protocolo USB, y los códigos PID son elegidos por cada fabricante por su cuenta. Por ejemplo, FTDI posee el VID 0403 y el Arduino Team, por su parte, también ha comprado su propio VID (2341) y lo utiliza con la placa Arduino UNO.

Cualquier sistema Linux (a partir del código fuente)

El código fuente del entorno de programación Arduino se encuentra disponible para descargar en http://github.com/arduino/Arduino. No obstante, esta

opción solamente se recomienda para usuarios avanzados; para instalar el IDE en un sistema Linux de forma rápida y cómoda se aconseja seguir los pasos descritos en los apartados anteriores. Si, aun así, se prefiere compilar el código fuente del IDE para adaptarlo mejor a las características de nuestro computador (o incluso para poder modificar personalmente alguna funcionalidad interna), en el siguiente enlace se pueden encontrar las instrucciones paso a paso para realizar con éxito el proceso: http://code.google.com/p/arduino/wiki/BuildingArduino.

Por otro lado, si se desea participar en el desarrollo del IDE, proponiendo mejoras o informando de errores, en http://code.google.com/p/arduino/issues/list podemos ver la lista de problemas en el código del IDE pendientes de resolver y su gestión; y en http://arduino.cc/pipermail/developers_arduino.cc podemos acceder a la lista de correo de los desarrolladores, donde se realizan consultas sobre el funcionamiento del código y se sugieren ideas para optimizarlo y enriquecerlo, además de discutir sobre el diseño de las diferentes placas y hardware oficial de Arduino.

Windows

Para instalar el IDE de Arduino en Windows 8, deberemos ir a su página web oficial de descargas: http://arduino.cc/en/Main/Software. Allí aparecerá, bajo el apartado "Downloads", un enlace para descargar la versión del IDE para Windows, que no es más que un archivo comprimido en formato zip. Por lo tanto, lo primero que tendremos que hacer una vez descargado es descomprimirlo. Cuando entremos dentro de la carpeta descomprimida que obtengamos, no encontraremos ningún instalador ni nada parecido (solo una estructura de archivos y subcarpetas que no deberemos modificar para nada) porque nuestro IDE ya está listo para ser usado desde este mismo momento: tan solo deberemos clicar sobre el archivo ejecutable "arduino.exe" y aparecerá ante nosotros.

Evidentemente, podremos mover y guardar en el lugar que deseemos de nuestro disco duro la carpeta contenedora de los ficheros que forman el IDE, siempre y cuando no alteremos nada de su interior. Seguramente también deseemos crear un acceso directo del ejecutable allí donde lo consideremos oportuno.

No obstante, no hemos acabado aún el proceso de instalación del IDE debido a que Windows 8 no incorpora de serie los "drivers" necesarios para poder reconocer la placa Arduino en el momento que esta se conecte mediante USB a nuestro computador. Así que para poder empezar a trabajar con nuestra placa debemos instalar previamente dichos drivers en nuestro sistema. Suponiendo que nuestra placa es el modelo Arduino UNO, los pasos a realizar son:

1. Conectar la placa Arduino UNO mediante USB a nuestro computador.

2. Acceder al “Administrador de dispositivos” de nuestro Windows 8. Para ello, debemos ubicarnos dentro del cuadro "Escritorio", y una vez allí, situar el cursor del ratón en la esquina superior derecha de la pantalla. Aparecerá un menú lateral de iconos; allí debemos seleccionar el icono "Configuración", y seguidamente, en el panel lateral recién surgido, debemos elegir la opción "Panel de control". Se abrirá una ventana en la que se muestran diferentes categorías; la que nos interesa es “Sistema y seguridad”. Clicando sobre ella podemos visualizar finalmente el enlace que nos lleva al "Administrador de dispositivos".

3. En la lista que nos muestra el "Administrador de dispositivos", debemos ver un “dispositivo desconocido”. Lo seleccionamos, clicamos con el botón derecho y elegimos la opción “Actualizar software de controlador”.

4. Elegimos “Buscar software de controlador en el equipo” y navegamos hasta la carpeta que contiene los ficheros “.inf”, los cuales incluyen la información necesaria para que Windows pueda finalmente instalar el driver correcto (hay un fichero “.inf” por cada modelo de placa Arduino). La carpeta donde están estos ficheros se llama “Drivers” y está situada dentro de la carpeta descomprimida que contiene los ficheros del IDE. Ojo, no confundir con la subcarpeta “FTDI USB Drivers”.

5. Hemos de hacer caso omiso de la advertencia por parte de Windows de la inexistencia de firma digital en los drivers de Arduino. Tras este paso, el proceso de instalación debería concluir sin mayor problema. Para comprobar que todo ha salido bien, deberíamos observar en la lista mostrada por el “Administrador de dispositivos” un nuevo dispositivo dentro de la categoría “Puertos (COM y LPT)” llamado “Arduino UNO (COMxx)” , donde “xx” será un número (normalmente será el 3 o el 4, pero puede ser otro). Este número será importante recordarlo posteriormente para poder utilizar nuestra placa correctamente.

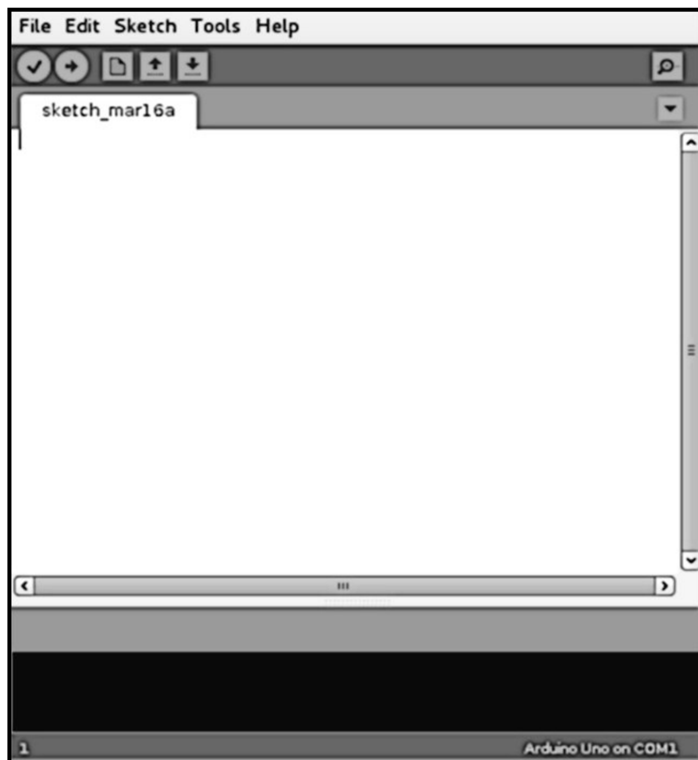
Mac OS X

Para instalar el IDE de Arduino en Mac OS X deberemos ir a su página web oficial de descargas: http://arduino.cc/en/Main/Software. Allí aparecerá, bajo el apartado “Downloads”, un enlace para descargar la versión del IDE para Mac OS X,

que no es más que un archivo comprimido en formato zip. Por lo tanto, lo primero que tendremos que hacer una vez descargado es descomprimirlo. Obtendremos entonces un archivo “dmg” ; haciendo doble clic sobre él comenzará el proceso de instalación estándar de cualquier aplicación nativa en sistemas Macintosh. El proceso es casi totalmente automático: lo único que deberemos hacer es, en la ventana que nos aparecerá al principio de la instalación, arrastrar el icono de Arduino sobre la carpeta Applications para que el IDE se instale allí (aunque de hecho podríamos elegir instalarlo en cualquier otro lugar). Y ya está: si todo va bien, en la barra de lanzadores aparecerá el icono correspondiente del recién instalado IDE de Arduino.

PRIMER CONTACTO CON EL IDE

Una vez instalado el IDE Arduino (en el sistema que sea), nada más arrancarlo veremos una ventana similar a la siguiente (la captura está realizada utilizando la versión 1.0.2 del IDE, que es la actual a fecha de edición de este libro):



Podemos ver que la ventana del IDE se divide en cinco grandes áreas. De arriba abajo son: la barra de menús, la barra de botones, el editor de código propiamente dicho, la barra y la consola de mensajes, y la barra de estado.

La zona del IDE donde trabajaremos más tiempo será el editor de código, ya que es allí donde escribiremos nuestros sketches. Otra zona que utilizaremos a menudo será la barra de botones, compuesta por los siguientes elementos:



Verify: este botón realiza dos cosas: comprueba que no haya ningún error en el código de nuestro sketch, y si el código es correcto, entonces lo compila. Este es el primer botón que deberemos pulsar cada vez que deseemos probar cualquier modificación que hagamos en nuestro sketch. (Si no sabes lo que significa “compilar”, no te preocupes: por ahora tan solo hay que saber que es un paso imprescindible).



Upload: este botón deberemos pulsarlo inmediatamente después del botón “Verify”. Su función es invocar internamente al comando “avrdude” para cargar en la memoria del microcontrolador de la placa Arduino el sketch recientemente verificado y compilado. “Avrdude” es capaz de realizar su tarea porque al hacer clic en este botón se activa automáticamente el bootloader del microcontrolador (el “auto-reset”), así que raras veces será necesario utilizar el método “tradicional” de activar el bootloader mediante la pulsación física del botón de reinicio que viene en la placa.



New: crea un nuevo sketch vacío.



Open: presenta un menú con todos los sketches disponibles para abrir. Podremos abrir tanto nuestros propios sketches como gran cantidad de sketches de ejemplo listos para probar, clasificados por categorías dentro del menú. Estos sketches son muy útiles para aprender; de hecho, en este texto haremos uso de bastantes de ellos, ya que son de dominio público.



Save: guarda el código de nuestro sketch en un fichero, el cual tendrá la extensión “.ino”. Podemos guardar estos ficheros donde queramos, pero el IDE Arduino nos ofrece una carpeta específica para ello, la carpeta “sketchbook”, ubicada dentro de la carpeta personal de nuestro usuario del sistema y generada automáticamente la primera vez que se ejecuta el IDE. En realidad, dentro de esta carpeta “sketchbook” se creará una subcarpeta diferente para cada proyecto dentro de la cual se guardarán los sketches correspondientes; de esta manera, sketches de diferentes proyectos no se mezclan entre sí.

 **Serial Monitor:** abre el "monitor serie". De él hablaremos enseguida.

Podemos ver también que justo debajo del botón de "Serial Monitor" tenemos un botón desplegable desde el cual podemos abrir nuevas pestañas. Tener varias pestañas abiertas a la vez nos puede ser útil cuando tenemos un código tan largo que necesitamos dividirlo en partes para trabajar más cómodamente. Esto es así porque todas las nuevas pestañas abiertas forman parte del mismo proyecto que la primera pestaña original (y por lo tanto, el código escrito en todas ellas es, en global, solo uno) pero el contenido particular de cada una de las pestañas se guarda físicamente en un fichero diferente, permitiendo así una manipulación más sencilla. En estos casos, cuando un proyecto consta de varios ficheros de código fuente, al abrir uno de ellos con el entorno de programación, este detectará la existencia de los demás ficheros incluidos en ese proyecto y los mostrará automáticamente en sus pestañas correspondientes. Lo más habitual es utilizar pestañas separadas para la definición de funciones, constantes o variables globales (conceptos que estudiaremos en el siguiente capítulo).

Otras acciones triviales que podemos realizar con el botón desplegable son cerrar la pestaña actual, renombrar la pestaña actual (lo que resulta en renombrar el sketch incluido en ella), moverse a la pestaña siguiente o anterior, o moverse a una pestaña concreta.

La barra de menú nos ofrece cinco entradas principales: "File", "Edit", "Sketch", "Tools" y "Help". Aclaremos que, posiblemente, estos nombres –y de hecho, todos los elementos del IDE– los verás traducidos ya de entrada al idioma configurado por defecto en tu sistema operativo, pero, para evitar posibles incoherencias, en este libro los seguiremos nombrando mediante su denominación anglosajona, que es la original. Las entradas de la barra de menú muestran acciones adicionales que complementan a las que podemos realizar con la barra de botones. Es interesante notar que los menús son sensibles al contexto, es decir, que solamente los elementos relevantes en un momento determinado (según lo que estemos haciendo) estarán disponibles, y los que no, aparecerán deshabilitados. Algunas de las acciones que podemos realizar mediante los menús son:

Menú "File": además de ofrecer acciones estándar como crear un nuevo sketch, abrir uno existente, guardarlo, cerrarlo, cerrar el IDE en sí, etc., podemos ver también otras acciones interesantes. Por ejemplo, gracias a la entrada "Examples" podemos acceder a los sketches de ejemplo que vienen de serie con el IDE y gracias a la entrada "Sketchbook" podemos acceder a nuestros propios sketches guardados en las diferentes subcarpetas que hay dentro de la carpeta "sketchbook". Otras acciones a tener en cuenta son por

ejemplo la de cargar el sketch en la memoria del microcontrolador (equivalente al botón “Upload” visto anteriormente), la de cargar el sketch en memoria utilizando un programador ISP externo (seleccionado previamente de entre la lista de programadores ISP compatibles, disponible en el menú “Tools”->”Programmer”) y la de imprimir el código del sketch por impresora (pudiendo configurar también el formato de la página). Finalmente, gracias a la entrada “Preferences”, podemos abrir un cuadro emergente que nos ofrece la posibilidad de establecer algunas preferencias del IDE, como por ejemplo la de cambiar la ubicación de la carpeta “sketchbook”, el idioma del IDE, el tamaño de la fuente de letra, la activación de las actualizaciones automáticas del IDE, el nivel de detalle en los mensajes mostrados durante el proceso de compilación y/o carga de los sketches, etc.

Existen muchas preferencias más que no aparecen en el cuadro emergente. Si queremos modificarlas, tendremos que hacerlo “a mano” editando el valor adecuado dentro del fichero de configuración “preferences.txt”, que no es más que un fichero de texto que contiene una lista de parejas dato<->valor bastante autoexplicativa. Dependiendo de nuestro sistema operativo, el fichero “preferences.txt” estará ubicado en una carpeta diferente, carpeta que ya se nos indica en el propio cuadro emergente. Sea como sea, este fichero se ha de modificar cuando el IDE no esté ejecutándose, porque si no, los cambios realizados serán sobrescritos por el propio entorno cuando se cierre.

Menú “Edit”: además de ofrecer acciones estándar como deshacer y rehacer, cortar, copiar y pegar texto, seleccionar todo el texto o buscar y reemplazar texto, podemos ver otras acciones interesantes. Por ejemplo, gracias a la entrada “Copy for forum” podemos copiar el código de nuestro sketch al portapapeles de nuestro sistema en una forma que es especialmente adecuada para pegarlo acto seguido directamente en el foro oficial de Arduino (y así poder recibir ayuda de la comunidad). Gracias a la entrada “Copy as HTML” podemos copiar el código de nuestro sketch al portapapeles de nuestro sistema en una forma que es especialmente adecuada para pegarlo en páginas web genéricas. Otras acciones a tener en cuenta son por ejemplo la de comentar/descomentar la porción de texto que tengamos seleccionada, o bien aplicarle o quitarle sangría (hablaremos de la importancia de estas dos posibilidades en el siguiente capítulo).

Menú “Sketch”: en este menú se ofrece la acción de verificar/compilar nuestro sketch (equivalente al botón “Verify” visto anteriormente), la de abrir la carpeta donde está guardado el fichero “.ino” que se esté editando

en este momento, la de añadir en una nueva pestaña un nuevo fichero de código a nuestro sketch y la de importar librerías.

Breve nota sobre cómo importar ("instalar") librerías:

Si nuestro sketch utiliza instrucciones pertenecientes a una librería determinada (ya sea oficial o bien de terceros), es necesario "importar" esa librería. Si se trata de una librería oficial, para ello tan solo debemos ir al menú "Sketch->Import library" y seleccionar la librería deseada del menú desplegable que aparece. Se puede observar que en hacer esto, al inicio del código de nuestro sketch se escribe automáticamente una (o más) líneas de apariencia similar a la siguiente: `#include <nombreLibreria.h>`

Esta línea (que también la podríamos haber escrito a mano perfectamente sin utilizar la opción del menú del IDE) permite utilizar en nuestro sketch las instrucciones pertenecientes a la librería indicada. Si esta línea no se escribe, la librería correspondiente no se importará e instrucciones pertenecientes a ella que aparezcan en el código del sketch no serán reconocidas, provocando un error de compilación.

Fijarse que (a diferencia del resto de instrucciones del lenguaje Arduino, tal como veremos en el próximo capítulo) no se escribe ningún punto y coma al final de la línea "#include". Esto es debido a que la palabra especial #include en realidad no es una instrucción sino que es lo que se llama "una directiva del preprocesador". Afortunadamente, no nos es necesario conocer este extremo: con recordar que al final de la línea #include no se ha de escribir punto y coma ya será suficiente.

La línea "#include" de una librería no oficial, en cambio, no es reconocida correctamente por el IDE Arduino hasta que no se instale esa librería en nuestro computador. Para ello, debemos descargarla de algún sitio web (normalmente en forma de archivo comprimido tipo "tgz"), debemos crear, si no existe ya, una subcarpeta llamada obligatoriamente "libraries" dentro de nuestra carpeta "sketchbook", y debemos descomprimir allí la librería en cuestión. Obtendremos una nueva carpeta, llamada igual que la librería, que alojará todos los ficheros que la forman. Automáticamente, la próxima vez que se inicie el IDE ya podremos importar esta nueva librería de la forma estándar. La razón de no poner las librerías de terceros en la misma carpeta donde están situadas las librerías oficiales (que es la carpeta llamada "libraries" y ubicada dentro del directorio de instalación del IDE) es para evitar que en una posible actualización del IDE, estas librerías extra sean eliminadas.

Los ficheros que forman una librería son básicamente ficheros con extensión .cpp (código fuente C/C++) y ficheros con extensión.h (cabeceras C/C++). Además, opcionalmente puede aparecer un fichero llamado "keywords.txt" (que indica al IDE Arduino qué nuevas palabras especiales ha de colorear cuando se escriban en nuestro sketch) y la carpeta "examples", que puede contener algunos sketches de ejemplo, los cuales aparecerán bajo el menú File->Examples. Queda claro que para poder programar nosotros una librería extra, es necesario que tengamos conocimientos relativamente avanzados del lenguaje de programación C y C++, así que este tema no lo abordaremos en este libro.

Hay que tener en cuenta que las librerías que utilicemos también se cargan en la memoria Flash del microcontrolador, por lo que aumentan el tamaño de nuestro sketch. Así que es buena idea eliminar las líneas #include correspondientes a librerías no utilizadas, si hubiera.

Menú "Tools": en este menú se ofrecen diferentes herramientas variadas, como la posibilidad de autoformatear el código para hacerlo más legible (por ejemplo, sangrando las líneas de texto contenidas dentro de las llaves { y }), la posibilidad de guardar una copia de todos los sketches del proyecto actual en formato zip, la posibilidad de abrir el monitor serie (equivalente al botón "Serial monitor"), etc. Otras herramientas mucho más avanzadas son por ejemplo la entrada "Programmer" ya comentada o la entrada "Burn bootloader", útil cuando queramos grabar un nuevo bootloader en el microcontrolador de la placa. Mención aparte requieren las entradas "Board" y "Serial", que comentaremos más extensamente en los párrafos siguientes.

Menú "Help": desde este menú podemos acceder a varias secciones de la página web oficial de Arduino que contienen diferentes artículos, tutoriales y ejemplos de ayuda. No se necesita Internet para consultar dichas secciones ya que esta documentación se descarga junto con el propio IDE, por lo que su acceso se realiza en local (es decir, "offline"). Se recomienda fervientemente su consulta.

La barra y la consola de mensajes informan en el momento de la compilación de los posibles errores cometidos en la escritura de nuestro sketch, además de indicar el estado en tiempo real de diferentes procesos, como por ejemplo la grabación de ficheros "ino" al disco duro, la compilación del sketch, la carga al microcontrolador, etc. Es interesante observar también que cada vez que se realice una compilación exitosa, aparecerá en la consola de mensajes el tamaño que ocuparía el sketch dentro de la memoria Flash del microcontrolador. Esta

información es muy valiosa, ya que si el tamaño de nuestro sketch estuviera próximo al tamaño máximo permitido (en el caso de la placa Arduino UNO recordemos que es de 32KB), podríamos saberlo y entonces modificar convenientemente el código de nuestro sketch para reducirlo de tamaño y no excedernos del límite.

La barra de estado simplemente muestra a su izquierda el número de línea del sketch actual donde en estos momentos está situado el cursor, y a su izquierda el tipo de placa Arduino y el puerto serie del computador actualmente utilizado.

El "Serial Monitor" y otros terminales serie

El "Serial monitor" es una ventana del IDE que nos permite desde nuestro computador enviar y recibir datos textuales a la placa Arduino usando el cable USB (más exactamente, mediante una conexión serie). Para enviar datos, simplemente hay que escribir el texto deseado en la caja de texto que aparece en su parte superior y clicar en el botón "Send" (o pulsar Enter). Aunque, evidentemente, no servirá de nada este envío si la placa no está programada con un sketch que sea capaz de obtener estos datos y procesarlos. Por otro lado, los datos recibidos provenientes de la placa serán mostrados en la sección central del "Serial monitor".

Es importante elegir mediante la caja desplegable de la parte inferior derecha del "Serial monitor" la misma velocidad de transmisión (en bits/s, también llamados "baudios") que la que se haya especificado en el sketch ejecutado en la placa, porque si no, los caracteres transferidos no serán reconocidos correctamente y la comunicación no tendrá sentido.

También es importante recordar que en sistemas MacOSX y Linux la placa Arduino UNO se autoresetea (es decir, recomienza a ejecutar el sketch desde el principio) cada vez que abrimos el "Serial monitor" y conectamos con ella. Esto con la placa Leonardo no ocurre.

No solo mediante el "Serial monitor" podemos comunicarnos mediante una conexión serie (a través de USB) con la placa. Se puede utilizar cualquier otro programa que permita enviar y recibir datos a través de conexiones de este tipo. Estos programas se suelen denominar "terminales serie". En los repositorios de Ubuntu y Fedora (y de la mayoría de distribuciones Linux) podemos encontrar varios, como "gtkterm", "cutecom", "picocom" o "minicom", entre otros. En Windows podemos utilizar el comando "telnet", que viene por defecto en una instalación estándar del sistema o programas de terceros como "putty" (http://www.putty.org) o "Terminalbpp" (https://sites.google.com/site/terminalbpp), entre otros. En Mac OS X una buena aplicación es "CoolTerm" (http://freeware.the-meiers.org).

En cualquiera de los programas anteriores no es necesario realizar ninguna configuración especial para que funcionen correctamente con nuestra placa: tan solo se requiere especificar la velocidad (en bits/s) a la que se efectuará la transmisión de datos y el puerto serie que se utilizará para establecer la comunicación (por ejemplo, /dev/ttyACM0 en Linux, COM3 en Windows, etc.; los valores posibles los discutiremos en el siguiente apartado), y ya está.

CONFIGURACIÓN Y COMPROBACIÓN DEL CORRECTO FUNCIONAMIENTO DEL IDE

Una vez hemos conectado mediante el cable USB nuestra placa recién adquirida a nuestro computador, lo primero que deberíamos ver es que el LED etiquetado como "ON" se enciende y se mantiene encendido de manera continua siempre. Una vez comprobado esto, podemos poner en marcha el entorno de programación Arduino. Una vez abierto, no obstante, antes de poder empezar a escribir una sola línea debemos asegurarnos de que el IDE "sepa" dos cosas básicas:

1. El tipo de placa Arduino conectada en este momento al computador (UNO, Leonardo, Mega, etc.)

2. El puerto serie de nuestro computador que ha de que utilizar para comunicarse vía USB con ella.

Para lo primero, simplemente debemos ir al menú "Tools"->"Boards" y seleccionar de la lista que aparece la placa con la que trabajaremos. Fijarse que hay modelos de placa que aparecen varias veces según si pueden venir con diferentes modelos de microcontrolador. Para lo segundo, debemos ir al menú "Tools"->"Serial port" y elegir el puerto serie (puede ser real o virtual) adecuado. Dependiendo del sistema operativo utilizado, en ese menú aparecerán opciones diferentes. Las más comunes son:

Sistemas Linux: aparecen los ficheros de dispositivo (generalmente /dev/ttyACM#).

Sistemas Windows: aparece una lista de puertos COM#, donde "#" es un número. Normalmente este número será 3 o superior, ya que los puertos COM1 y COM2 se suelen reservar habitualmente para puertos serie hardware reales (no simulados a través de USB). Si aparecen varios puertos COM y no se sabe cuál es el correspondiente a la placa Arduino, lo más

sencillo es desconectarla y reabrir el menú: la entrada que haya desaparecido será la que buscamos; conectando otra vez la placa podremos seleccionarla sin problemas.

Sistemas MacOS X: aparecen los ficheros de dispositivo (normalmente /dev/tty.usbmodem###). Es posible que en el momento de conectar una placa UNO o Mega se muestre una ventana emergente informando de que se ha detectado un nuevo dispositivo de red. Si ocurre esto, basta con clicar en "Network preferences..." y cuando aparezca la nueva ventana, simplemente clicar en "Apply". La placa se mostrará como "Not configured", pero funcionará correctamente por lo que ya se puede salir de las preferencias del sistema.

Ambas configuraciones (placa y puerto) solamente serán necesarias especificarlas una sola vez mientras no las modifiquemos (es decir, mientras no usemos otro tipo de placa, por ejemplo).

Ahora solo nos falta comprobar que todo funcione correctamente, ejecutando algún sketch de prueba. Como todavía no sabemos escribir ninguno por nuestra cuenta, utilizaremos un sketch de los que vienen como ejemplo. Concretamente, el sketch llamado "Blink". Para abrirlo, debemos ir al menú "File"->"Examples"->"01.Basics"->"Blink". Ahora no nos interesa estudiar el código que aparece: simplemente hemos de pulsar en el botón "Verify" y seguidamente en el botón "Upload". Siempre deberemos seguir esta pauta con todos los sketches: una vez queramos probar cómo funcionan, debemos pulsar primero en "Verify" (para compilar y verificar el código) y seguidamente en "Upload" (para cargarlo en el microcontrolador). Justo después de haber pulsado este último botón, ocurre que:

1. Primero parpadeará muy rápidamente el LED etiquetado como "L", indicando que la placa se ha reseteado y, por lo tanto, que se está ejecutando el bootloader

2. Seguidamente los LEDs de la placa etiquetados como "RX" y "TX" parpadearán rápidamente varias veces, indicando que el sketch está llegando a la placa y, por tanto, que está siendo recibido por el bootloader y cargado en la memoria Flash del microcontrolador. En el IDE deberemos observar mensajes informando del estado del proceso, y de su finalización exitosa.

3. Finalmente, una vez acabada la carga, lo que debería ocurrir es que el LED etiquetado como "L" empezara a parpadear de forma periódica, ya para

siempre. Si ocurre eso, ¡felicidades!: nuestra placa funciona perfectamente y nuestro computador la puede programar sin problemas.

MÁS ALLÁ DEL LENGUAJE ARDUINO: EL LENGUAJE C/C++

A lo largo de los párrafos anteriores hemos repetido varias veces la palabra "compilación", pero no hemos explicado su significado hasta ahora. "Compilar" significa convertir un código escrito en el IDE (utilizando el lenguaje Arduino, y por tanto, entendible fácilmente por los seres humanos) en el programa realmente ejecutable por el microcontrolador, que no es más que un inmenso conjunto de bits (es decir, 1s y 0s) tan solo entendible por él.

Es decir, nosotros escribimos en el IDE las instrucciones usando el sencillo lenguaje de programación Arduino y posteriormente, mediante la compilación, transformamos ese sketch en instrucciones "digeribles" para el microcontrolador (en lo que viene a llamarse "código máquina" o "código binario"). Este código máquina en esencia no es más que un conjunto de impulsos eléctricos (1s –pasa corriente– y 0s –no pasa corriente–), que es lo único que realmente saben procesar los circuitos electrónicos. Así pues, un código máquina (ficticio) resultante de la compilación de un sketch Arduino cualquiera podría ser similar a esto: 1001011101010101011011...

Es evidente que es imposible escribir un programa directamente en código máquina: por eso existen los compiladores. Pero, además, estas herramientas ofrecen otra ventaja: hay que saber que el código máquina válido para un microcontrolador no lo es para otro, debido a su diferente construcción electrónica interna. Por lo tanto, un mismo programa se tendría que codificar en diferentes códigos máquina para diferentes modelos de microcontrolador. Sin embargo, si disponemos de compiladores específicos para cada uno de estos modelos, a partir de un mismo código fuente podemos obtener diferentes códigos máquina utilizando en cada caso el compilador respectivo. Esto es una gran ventaja, porque nos permite no reescribir código ya funcional ni cometer errores en nuestros desarrollos, además de aportar una gran flexibilidad y escalabilidad a nuestros sketches.

Ya que las placas Arduino incorporan microcontroladores de arquitectura AVR (excepto la Due, que es ARM, pero no lo tendremos en cuenta), es lógico pensar que el compilador incluido en el entorno de programación oficial de Arduino es uno específico para generar código binario compatible con este tipo de chips. Y así es. Sin embargo, dicha herramienta (llamada "gcc-avr") no compila código escrito en lenguaje Arduino a código binario AVR (concretamente, en formato ".hex"), sino que compila código escrito en lenguaje C/C++ a código binario AVR. ¿Por qué?

Porque realmente, el lenguaje Arduino no es un lenguaje en sentido estricto: es simplemente un conjunto de instrucciones C y C++ "camufladas", diseñadas para simplificar el desarrollo de programas para microcontroladores AVR. Es decir, cuando estamos escribiendo nuestro sketch en "lenguaje Arduino", sin saberlo en realidad estamos programando en una versión simplificada del lenguaje C/C++. Si se desea conocer en detalle todo el proceso interno de transformación del sketch originalmente escrito en lenguaje Arduino a su versión en lenguaje C++ y su posterior compilación por "gcc-avr", en http://code.google.com/p/arduino/wiki/BuildProcess se puede consultar la información necesaria.

El lenguaje C y su "pariente" C++ son dos de los lenguajes más importantes y extendidos del mundo por varias razones: porque son lenguajes muy potentes y a la vez ligeros y flexibles, porque poseen un ecosistema amplísimo de librerías que los dotan de funcionalidad que otros lenguajes no ofrecen, porque los programas escritos y compilados en estos lenguajes son tremendamente eficientes y rápidos, y porque existen compiladores para prácticamente cualquier tipo de hardware (con lo que hoy en día podemos ver multitud de software escrito con estos lenguajes ejecutándose en una gran variedad de máquinas).

No obstante, tal vez para una persona que se inicia en el mundo de la programación, los lenguajes C/C++ no son demasiado amigables. En otras palabras: son lenguajes relativamente difíciles de aprender y dominar. Por eso existe el lenguaje Arduino: para que una persona sin apenas conocimientos de desarrollo de aplicaciones pueda escribir rápidamente un sketch Arduino funcional sin tener que aprender todo un lenguaje de programación completo pero complejo como es C o C++. Es decir: el lenguaje Arduino hace de "máscara" del lenguaje C/C++, ocultando gran cantidad de detalles superfluos para el entusiasta de los proyectos electrónicos y facilitando así el uso de la plataforma Arduino en conjunto. El "precio" a pagar por ganar esta facilidad en la programación de sketches es que tan solo tenemos un subconjunto de toda la funcionalidad que puede llegar a ofrecer el lenguaje C/C++. Afortunadamente, para la mayoría de proyectos no es necesario ir más allá de lo que ya nos ofrece el lenguaje Arduino y su IDE oficial, y por supuesto, todos los ejemplos de este libro están programados utilizando solo las funcionalidades que ofrece el lenguaje Arduino en exclusiva.

IDEs ALTERNATIVOS AL OFICIAL

Hay gente por todo el mundo a la que el IDE oficial de Arduino no le acaba de convencer, por varios motivos: la falta de funcionalidades avanzadas (como el

autocompletado de sentencias, por ejemplo) o la dependencia del lenguaje Java (hecho que implica tener instalado en nuestro computador más paquetes de los realmente necesarios para poder compilar y cargar nuestros sketches) son algunos. Otra razón es que muchos programadores ya están familiarizados con un IDE concreto y quieren seguir utilizándolo para sus desarrollos con Arduino. La consecuencia de esto es que actualmente existe una variedad de IDEs "alternativos" que aportan más características que el IDE original o simplemente que cambian la manera de trabajar. A continuación, se nombran algunos de estos IDEs (¡no es una lista exhaustiva!) por si el lector quiere buscar en ellos alguna característica que el IDE oficial no tiene:

CodeBlocks (http://www.codeblocks.org): es un IDE libre y multiplataforma para el desarrollo de aplicaciones escritas en lenguaje C y C++. No obstante, se pueden escribir programas directamente en lenguaje Arduino (y también cargarlos en la placa) mediante una versión modificada de este software, la llamada "CodeBlocks Arduino Edition", descargable desde la dirección http://www.arduinodev.com/codeblocks.

Gnoduino (http://gnome.eu.org): IDE libre que intenta imitar en aspecto y funcionalidad al oficial, pero que evita su dependencia de Java, ya que está escrito en otro lenguaje, Python. Esto hace que sea especialmente ligero y rápido. No obstante, solo funciona en Linux, y más específicamente dentro el escritorio GNOME debido a sus dependencias.

Codebender (http://www.codebender.cc): este IDE en realidad es una aplicación web, por lo que funciona enteramente de forma "online" dentro del navegador, sin necesidad de instalar nada. Incluye un completo editor de textos, un compilador y un cargador de sketches, todo sin salir del navegador. Además, previo registro gratuito, permite almacenar "en la nube" el conjunto de códigos realizados. Este proyecto es libre, por lo que uno se puede descargar e instalar el software necesario para montar una plataforma Codebender propia.

Visualmicro (http://visualmicro.codeplex.com): en realidad se trata de un complemento ("plug-in") para usar el lenguaje Arduino dentro del entorno de programación Visual Studio (http://www.microsoft.com/visualstudio), de Microsoft. Desgraciadamente, no funciona en la versión Express de este entorno (la gratuita), por lo que se ha de comprar alguna edición superior, como por ejemplo la Professional.

EmbedXcode (http://embedxcode.weebly.com): en realidad se trata de un complemento ("plug-in") del entorno de programación oficial de Apple, llamado XCode (https://developer.apple.com/xcode). Permite trabajar con un IDE "todo en uno" mediante el cual se pueden programar de forma unificada diversas plataformas, como Arduino, chipKIT, Maple, MSP430 o Wiring, (aunque cada una mediante su lenguaje de programación propio).

De todas formas, si no queremos utilizar ninguno de los entornos de desarrollo compatibles anteriores, aún tenemos la posibilidad de escribir código Arduino con absolutamente cualquier otro IDE que queramos. Para ello, deberemos tener instalado tanto nuestro IDE favorito como el entorno oficial de Arduino, y seguir los pasos indicados a continuación. De esta forma lograremos utilizar todas las funcionalidades para la edición de código que ofrezca IDE favorito y utilizar el entorno oficial solo para compilar y cargar.

1. Ejecutar el IDE Arduino y abrir el fichero ".ino" que deseemos editar.

2. Ir al cuadro de preferencias del IDE y activar la opción "Use external editor". Automáticamente, el editor de código aparecerá en color gris, indicando que está deshabilitado.

3. Ejecutar el IDE que deseemos, y abrir el mismo fichero ".ino". Realizar la edición necesaria.

4. Grabar los cambios en el editor utilizado. Es importante que el fichero ".ino" grabado mantenga el mismo nombre que el fichero ".ino" originalmente abierto en el IDE Arduino, porque si no este no se enterará.

5. Utilizar los botones de "Verify" y "Upload" del IDE Arduino de la forma habitual, cuando se considere oportuno.

Por otro lado, conviene destacar también la existencia de un tipo de entornos de desarrollo para Arduino un tanto "especiales", ya que están enfocados a la programación visual de sketches. Es decir: en vez de utilizar instrucciones escritas en un lenguaje abstracto, los códigos se construyen a partir del "acoplamiento" gráfico de diferentes bloques coloreados que representan acciones y estructuras de control. Su objetivo es facilitar a cualquier persona sin ninguna experiencia en programación (por ejemplo, los niños) la iniciación en el mundo de los microcontroladores, matando de esta forma dos pájaros de un tiro: la introducción a la programación y la introducción a la electrónica. Los más destacables son:

Scratch for Arduino -S4A- (http://seaside.citilab.eu/scratch/arduino): es un plug-in del entorno visual de programación Scratch (http://scratch.mit.edu) que permite usar este para programar e interactuar con placas Arduino.

Modkit Micro (http://www.modk.it): entorno de desarrollo visual muy similar a Scratch. Puede utilizarse en más arquitecturas además de las placas Arduino, y tiene la particularidad de poder funcionar "online" dentro del navegador. También se puede descargar el entorno instalable, pero solo para Windows y MacOSX. Otra característica es que si se desea, se puede trabajar con el código interno más allá de los bloques visuales, permitiendo una personalización más directa de los sketches.

Minibloq (http://blog.minibloq.org): igual que el anterior, es un entorno visual de programación listo para ser usado en más arquitecturas que Arduino solamente, pero solamente funciona en Windows.

Ardublock (http://blog.ardublock.com): entorno de desarrollo visual para Arduino programado en Java (y por tanto, multiplataforma).

Destacar finalmente, aunque no sea un entorno de desarrollo propiamente dicho, la herramienta **Ino** (http://inotool.org). Se trata de un programa ejecutable directamente desde el intérprete de comandos que permite compilar y cargar sketches Arduino. Estos sketches, eso sí, deben haber sido escritos y guardados previamente utilizando un editor de texto cualquiera. Incorpora además una opción equivalente al "Serial monitor". Sus detalles de configuración (como por ejemplo el modelo de placa o el puerto serie a usar en cada momento) se pueden especificar como parámetros del comando o bien dentro de un archivo de configuración específico. Se recomienda consultar la documentación existente en su página web oficial para aprender su uso. Desgraciadamente, no funciona en sistemas Windows.

4 LENGUAJE ARDUINO

MI PRIMER SKETCH ARDUINO

Conecta la placa Arduino a tu computador y ejecuta el IDE oficial. Selecciona (si no lo está ya) el tipo de placa adecuado (en el menú Tools->Board) y el puerto USB utilizado (en el menú Tools->Serial port).

Ejemplo 4.1: Crea un nuevo sketch con el siguiente contenido.

```
/*Declaración e inicialización de una
  variable global llamada "mivariable" */
int mivariable=555;
void setup() {
       Serial.begin(9600);
}
void loop() {
       Serial.println(mivariable);
       mivariable=mivariable+1;
}
```

Pulsa en el botón “Verify” y seguidamente en el botón “Upload”. No deberías de observar ningún error en la consola de mensajes. Abre ahora el “Serial monitor” y verás que allí van apareciendo en tiempo real muchos números uno tras otro, empezando por el 555 y siguiendo por el 556, 557, 558, 559, etc., aumentando sin parar. ¿Por qué? ¿Qué significa este texto (este código) que hemos introducido en la memoria del microcontrolador de la placa?

ESTRUCTURA GENERAL DE UN SKETCH

Un programa diseñado para ejecutarse sobre un Arduino (un "sketch") siempre se compone de tres secciones:

La sección de declaraciones de variables globales: ubicada directamente al principio del sketch.

La sección llamada "void setup()": delimitada por llaves de apertura y cierre.

La sección llamada "void loop()": delimitada por llaves de apertura y cierre.

La primera sección del sketch (que no tiene ningún tipo de símbolo delimitador de inicio o de final) está reservada para escribir, tal como su nombre indica, las diferentes declaraciones de variables que necesitemos. En un apartado posterior explicaremos ampliamente qué significa todo esto.

En el interior de las otras dos secciones (es decir, dentro de sus llaves) deberemos escribir las instrucciones que deseemos ejecutar en nuestra placa, teniendo en cuenta lo siguiente:

Las instrucciones escritas dentro de la sección "void setup()" se ejecutan una única vez, en el momento de encender (o resetear) la placa Arduino.

Las instrucciones escritas dentro de la sección "void loop()" se ejecutan justo después de las de la sección "void setup()" infinitas veces hasta que la placa se apague (o se resetee). Es decir, el contenido de "void loop()" se ejecuta desde la 1ª instrucción hasta la última, para seguidamente volver a ejecutarse desde la 1ª instrucción hasta la última, para seguidamente ejecutarse desde la 1ª instrucción hasta la última, y así una y otra vez.

Por tanto, las instrucciones escritas en la sección "void setup()" normalmente sirven para realizar ciertas preconfiguraciones iniciales y las instrucciones del interior de "void loop()" son, de hecho, el programa en sí que está funcionando continuamente.

En el caso concreto del ejemplo 4.1, vemos que en la zona de declaraciones de variables globales hay una sola línea (`int mivariable=555;`) que dentro de "void setup()" se ejecuta una sola instrucción (`Serial.begin(9600);`) y que dentro de "void loop()" se realiza la ejecución continua y repetida (hasta que la alimentación de la placa se interrumpa) de dos instrucciones una tras otra:

`Serial.println(mivariable);` y `mivariable=mivariable+1;`. Sobre el significado y sintaxis de todas estas líneas hablaremos a continuación.

Sobre las mayúsculas, tabulaciones y puntos y comas

Conviene aclarar ya pequeños detalles que deberemos tener en cuenta a la hora de escribir nuestros sketches para evitarnos muchos dolores de cabeza. Por ejemplo, es necesario saber que el lenguaje Arduino es "case-sensitive". Esto quiere decir que es totalmente diferente escribir una letra en mayúscula que en minúscula. Dicho de otra forma: para el lenguaje Arduino "HolA" y "hOLa" son dos palabras distintas. Esto tiene una implicación muy importante: no es lo mismo escribir por ejemplo "Serial.begin(9600);" que "serial.begin(9600);". En el primer caso la instrucción estaría correctamente escrita, pero en el segundo, en el momento de compilar el código el IDE se quejaría porque para él "serial" (con "s" minúscula) no tiene ningún sentido. Así que hay que vigilar mucho con respetar esta distinción en los códigos que escribamos.

Otro detalle: las tabulaciones de las instrucciones contenidas dentro de las secciones "void setup()" y "void loop()" del sketch del ejemplo 4.1 no son en absoluto necesarias para que la compilación del sketch se produzca con éxito. Simplemente son una manera de escribir el código de forma ordenada, clara y cómoda para el programador, facilitándole la tarea de leer código ya escrito y mantener una cierta estructura a la hora de escribirlo. En los próximos ejemplos de este libro se irá viendo mejor su utilidad.

Otro detalle: todas las instrucciones (incluyendo también las declaraciones de variables) acaban con un punto y coma. Es indispensable añadir siempre este signo para no tener errores de compilación, ya que el compilador necesita localizarlo para poder detectar el final de cada instrucción escrita en el sketch. Si se olvida, se mostrará un texto de error que puede ser obvio ("falta un punto y coma") o no. Si el texto del error es muy oscuro o sin lógica, es buena idea comprobar que la causa no sea la falta de un punto y coma en las líneas justamente anteriores a la marcada por el compilador como causante del problema.

COMENTARIOS

La primera línea del sketch del ejemplo 4.1 contiene un comentario (concretamente, son las dos primeras líneas: desde los símbolos /* hasta los símbolos */). Un "comentario" es un texto escrito intercalado con el código del sketch que se

utiliza para informar sobre cómo funciona ese código a la persona que en algún momento lo esté leyendo. Es decir, los comentarios son texto de ayuda para los seres humanos que explica el código asociado y ayudan a entenderlo y recordar su función. Los comentarios son completamente ignorados y desechados por el compilador, por lo que no forman parte nunca del código binario que ejecuta el microcontrolador (así que no ocupan espacio en su memoria).

Los comentarios pueden aparecer dentro del código de diferentes formas:

Comentarios compuestos por una línea entera (o parte de ella): para añadirlos deberemos escribir dos barras (//) al principio de cada línea que queramos convertir en comentario. También podemos comentar solamente una parte de la línea, si escribimos las barras en otro punto que no sea el principio de esta; de esta manera solamente estaremos comentando lo que aparece detrás de las barras hasta el final de la línea, pero lo anterior no.

Comentarios compuestos por un bloque de varias líneas seguidas: para añadirlos deberemos escribir una barra seguida de un asterisco (/*) al principio del bloque de texto que queramos convertir en comentario, y un asterisco seguido de una barra (*/) al final de dicho bloque. Todos los caracteres y líneas ubicados entre estas dos marcas de inicio y final serán tratadas automáticamente como comentarios. Hay que tener en cuenta, por otro lado, que comentarios unilineales se pueden escribir dentro un comentario multilineal, pero uno multilineal dentro de otro no. Este es el tipo de comentario escrito en el sketch del ejemplo 4.1.

Una práctica bastante habitual en programación es comentar en algún momento una (o más) partes del código. De esta forma, se "borran" esas partes (es decir, se ignoran, y por tanto, ni se compilan ni se ejecutan) sin borrarlas realmente. Normalmente, esto se hace para localizar posibles errores en el código observando el comportamiento del programa con esas determinadas líneas comentadas. A lo largo de los ejemplos de este libro se irá viendo su utilidad.

VARIABLES

La primera línea del sketch del ejemplo 4.1 consiste en declarar una variable global de tipo "int" llamada "mivariable", e inicializarla con el valor de 555. Expliquemos todo esto.

Una variable es un elemento de nuestro sketch que actúa como un pequeño "cajoncito" (identificado por un nombre elegido por nosotros) que guarda un determinado contenido. Ese contenido (lo que se llama el valor de la variable) se podrá modificar en cualquier momento de la ejecución del sketch: de ahí el nombre de "variable". La importancia de las variables es inmenso, ya que todos los sketches hacen uso de ellas para alojar los valores que necesitan para funcionar.

El valor de una variable puede haberse obtenido de diversas maneras: puede haber sido asignado literalmente (como el del ejemplo 4.1, donde nada más empezar asignamos explícitamente a la variable llamada "mivariable" el valor 555), pero también puede ser el dato obtenido por un sensor, o el resultado de un cálculo, etc. Provenga de donde provenga inicialmente, ese valor podrá ser siempre cambiado en cualquier instante posterior de la ejecución del sketch, si así lo deseamos.

Declaración e inicialización de una variable

Antes de poder empezar a utilizar cualquier variable en nuestro sketch; no obstante, primero deberemos crearla. Al hecho de crear una variable se le suele llamar "declarar una variable". En el lenguaje Arduino, cuando se declara una variable es imprescindible además especificar su tipo. El tipo de una variable lo elegiremos según el tipo de datos (números enteros, números decimales, cadena de caracteres, etc.) que queramos almacenar en esa variable. Es decir: cada variable puede guardar solamente valores de un determinado tipo, por lo que deberemos decidir en su declaración qué tipo de variable es la que nos interesa más según el tipo de datos que preveamos almacenar. Asignar un valor a una variable que sea de un tipo diferente al previsto para esta provoca un error del sketch.

Los tipos posibles del lenguaje Arduino los detallaremos en los párrafos siguientes, pero ya se puede saber que la sintaxis general de una declaración es siempre una línea escrita así: *tipoVariable nombreVariable;*. En el caso concreto de querer declarar varias variables del mismo tipo, en vez de escribir una declaración por separado para cada una de ellas, es posible declararlas todas en una misma línea, genéricamente así: *tipoVariable nombreVariable1, nombreVariable2;*.

Opcionalmente, a la vez que se declara la variable, se le puede establecer un valor inicial: a esto se le llama "inicializar una variable". Inicializar una variable cuando se declara no es obligatorio, pero sí muy recomendable. En el ejemplo 4.1, declaramos la variable llamada "mivariable" de tipo "int" (que es un tipo de dato de entre los varios existentes pensados para guardar números enteros) y además, también la inicializamos con el valor inicial de 555. De ahí ya se puede deducir que la sintaxis general de una inicialización es siempre una línea escrita así: *tipoVariable nombreVariable = valorInicialVariable;*.

Uno podría preguntarse por qué es conveniente usar variables inicializadas en vez de escribir su valor directamente allí donde sea necesario. Por ejemplo, en el caso del código del ejemplo 4.1, hemos asignado el valor de 555 a “mivariable” cuando lo podríamos haber escrito directamente dentro de la instrucción Serial.println() así: `Serial.println(555);`. La razón principal es porque trabajar con variables nos facilita mucho la comprensión y el mantenimiento de nuestros programas: si ese valor aparece escrito en multitud de líneas diferentes de nuestro código y ha de ser cambiado, utilizando una variable inicializada con ese valor tan solo deberíamos cambiar la inicialización de la variable y automáticamente todas las veces donde apareciera esa variable dentro de nuestro código tendría este nuevo valor; si escribiéramos directamente ese valor en cada línea, tendríamos que irlo modificando línea por línea, con la consecuente pérdida de tiempo y la posibilidad de cometer errores.

Por otro lado, a la hora de declarar las variables, se recomienda dar a nuestras variables nombres descriptivos para hacer el código de nuestro sketch más legible. Por ejemplo, nombres como “sensorDistancia” o “botonEncendido” ayudarán a entender mejor lo que esas variables representan. Para nombrar una variable se puede utilizar cualquier palabra que queramos, siempre que esta no sea ya una palabra reservada del lenguaje Arduino (como el nombre de una instrucción, etc.) y que no empiece por un dígito.

Asignación de valores a una variable

¿Qué pasa si una variable se declara pero no se inicializa? Tendrá un valor por defecto (normalmente sin interés para nosotros) hasta que se le asigne un valor diferente en algún momento de la ejecución de nuestro sketch. La sintaxis general para asignar un nuevo valor a una variable (ya sea porque no se ha inicializado, o sobre todo porque se desea sobrescribir un valor anterior por otro nuevo), es: *nombreVariable = nuevoValor;.*

Por ejemplo, si la variable se llama “mivariable” y es de tipo entero, podría escribirse en el interior de alguna sección de nuestro sketch una línea tal como `mivariable=567;` para asignarle el nuevo valor numérico de 567. Importante fijarse en que la línea de asignación se lee “de izquierda a derecha”: el valor “nuevoValor” se asigna a la variable “nombreVariable”.

Las asignaciones de valores a variables pueden ser muy variadas: no siempre son valores directos como en el ejemplo 4.1. Un caso bastante habitual es asignar a una variable un valor que depende del valor de otra. Por ejemplo, si suponemos que tenemos una variable llamada “y” y otra llamada “x”, podríamos escribir lo siguiente:

`y = x + 10;`. La sentencia anterior se ha de leer así: al valor que tenga actualmente la variable “x” se le suma 10 y el resultado se asigna a la variable “y” (es decir, que si por ejemplo “x” tiene un valor de 5, “y” tendrá un valor de 15).

Incluso nos podemos encontrar con asignaciones del tipo `y = y + 1;` (como de hecho ocurre con el sketch del ejemplo 4.1). La clave aquí está en entender que el símbolo “=” no es el de la igualdad matemática (que ya estudiaremos) sino el de la asignación. Por tanto, una línea como la anterior lo que hace es sumar una unidad al valor actual de la variable “y” (se entiende que es de tipo numérico) para seguidamente asignar este nuevo valor resultante otra vez a la variable “y”, sobrescribiendo el que tenía anteriormente. Es decir, que si inicialmente “y” vale 45 (por ejemplo), después de ejecutarse la asignación `y = y + 1;` valdrá 46.

Ámbito de una variable

Otro concepto importante en relación con las variables es el de ámbito de una variable. En este sentido, una variable puede ser “global” o “local”. Que sea de un ámbito o de otro depende de en qué lugar de nuestro sketch se declare la variable:

Para que una variable sea global se ha de declarar al principio de nuestro sketch; es decir, antes (y fuera) de las secciones “void setup()” y “void loop()”. De hecho, al hablar de la estructura de un sketch ya habíamos mencionado la existencia de esta sección de declaraciones de variables globales. Una variable global es aquella que puede ser utilizada y manipulada desde cualquier punto del sketch. Es decir, todas las instrucciones de nuestro programa, sin importar dentro de qué sección estén escritas (“void setup()”, “void loop()” u otras que puedan existir) pueden consultar y también cambiar el valor de dicha variable.

Para que una variable sea local se ha de declarar en el interior de alguna de las secciones de nuestro sketch (es decir, dentro de “void setup()” o de “void loop()” o de otras que puedan existir). Una variable local es aquella que solo puede ser utilizada y manipulada por las instrucciones escritas dentro de la misma sección donde se ha declarado. Este tipo de variables es útil es sketches largos y complejos para asegurarse de que solo una sección tiene acceso a sus propias variables, ya que esto evita posibles errores cuando una sección modifica inadvertidamente variables utilizadas por otra sección.

Tipos posibles de una variable

Los tipos de variables que el lenguaje Arduino admite son:

El tipo “boolean”: las variables de este tipo solo pueden tener dos valores: cierto o falso. Se utilizan para almacenar un estado de entre esos dos posibles, y así hacer que el skech reaccione según detecte en ellas uno u otro. Por ejemplo, las variables booleanas se pueden usar para comprobar si se han recibido datos de un sensor (cierto) o no (falso), para comprobar si algún actuador está disponible (cierto) o no (falso), para comprobar si el valor de otra variable diferente cumple una determinada condición como por ejemplo la de ser mayor que un número concreto (cierto) o no (falso). El valor guardado en una variable booleana ocupa siempre un byte de memoria.

Para asignar explícitamente a una variable de tipo “boolean” el valor de cierto, se puede utilizar la palabra especial “true” (sin comillas) o bien el valor “1” (sin comillas), y para asignarle el valor de falso se puede utilizar la palabra especial “false” (sin comillas) o bien el valor “0” (sin comillas). Es decir, si en nuestro sketch de ejemplo la variable “mivariable” hubiera sido de tipo “boolean” en vez de “int”, para asignarle el valor de “true” tendríamos que haber escrito una línea similar a `boolean mivariable=true;` o bien `boolean mivariable=1;` (en realidad, una variable booleana con un valor cualquiera diferente de 0 ya se interpreta que tiene un valor cierto: no tiene por qué ser el valor 1 concretamente).

El tipo “char”: el valor que puede tener una variable de este tipo es siempre un solo carácter (una letra, un dígito, un signo de puntuación...). Si lo que queremos es almacenar una cadena de caracteres (es decir, una palabra o una frase) el tipo “char” no nos sirve, deberemos usar otro tipo explicado posteriormente.

Para asignar explícitamente a una variable de tipo “char” un determinado valor (es decir, un carácter), deberemos tener la precaución de escribir ese carácter entre comillas simples. Por tanto, si en el sketch del ejemplo 4.1 la variable “mivariable” hubiera sido de tipo “char” en vez de “int”, para asignarle el valor de la letra A tendríamos que haber escrito una línea similar a `char mivariable='A';`.

En realidad, los caracteres se almacenan internamente como números ya que los dispositivos electrónicos son incapaces de trabajar con “letras” directamente: las han de “traducir” siempre primero a números para entonces poderlas almacenar y procesar. Para saber a qué número interno corresponde un determinado carácter, y

viceversa, la placa Arduino utiliza la llamada tabla ASCII, la cual es una simple lista de equivalencias que asocia cada carácter con un número determinado.

El hecho de que los caracteres para la placa Arduino en realidad sean reconocidos como números permite que sea posible realizar operaciones aritméticas con esos caracteres (o mejor dicho, con su valor numérico correspondiente dentro de la tabla ASCII). Por ejemplo: si realizamos la operación 'A' + 1 obtendríamos el valor 66, ya que en la tabla ASCII el valor numérico del carácter 'A' es 65. De hecho, incluso podríamos inicializar una variable "char" asignándole un valor numérico en vez del carácter correspondiente (es decir: la línea `char mivariable='A';` la podríamos escribir como `char mivariable=65;` y ambas serían equivalentes).

Cada variable de tipo carácter ocupa 8 bits (un byte) de memoria para almacenar su valor. Esto implica que existen $2^8 = 256$ valores diferentes posibles para una variable de este tipo (es fácil ver esto si se cuentan las combinaciones de 0s y 1s que se pueden obtener con 8 posiciones). Como los valores de tipo "char" en realidad son números, los valores que puede almacenar una variable de este tipo están dentro un rango que va desde el número -128 hasta el 127.

Breve nota sobre la tabla ASCII:

La tabla ASCII original (luego le han sucedido extensiones) tan solo lista 128 caracteres con sus correspondientes equivalencias numéricas (las cuales van de 0 a 127). Esto es así porque se utilizaron 7 bits para codificar los caracteres, y con 7 bits solo se pueden utilizar este número de combinaciones ($2^7 = 128$).

Los 32 primeros números de la tabla ASCII, utilizados originalmente para gestionar dispositivos tales como impresoras a través de terminales remotos, se denominan "códigos de control" y son caracteres no imprimibles. De ellos, los más importantes para nosotros actualmente son el nº 0 (carácter nulo), el nº 8 (tecla de retroceso), el nº 9 (tabulador), el nº 10 (salto de línea), el nº 13 (retorno de carro), el nº 27 (escape) o el nº 32 (espacio).

Los caracteres ASCII imprimibles (a partir del 33) se pueden visualizar en la pantalla de un computador PC cualquiera pulsando el código numérico correspondiente en el teclado numérico (para ello la tecla NumLock ha de estar activa) mientras se mantiene pulsada la tecla ALT -GR. El apéndice B muestra estos códigos como referencia para el lector.

El tipo "byte": el valor que puede tener una variable de este tipo es siempre un número entero entre 0 y 255. Al igual que las variables de tipo "char", las de tipo "byte" utilizan un byte (8 bits) para almacenar su valor y, por tanto, tienen el mismo número de combinaciones numéricas posibles diferentes (256), pero a diferencia de aquellas, los valores de una variable "byte" no pueden ser negativos.

El tipo "int": el valor que puede tener una variable de este tipo es un número entero entre -32768 (-2^{15}) y 32767 ($2^{15}-1$), gracias a que utilizan 2 bytes (16 bits) de memoria para almacenarse. Esto es así para todas las placas Arduino excepto para la Due: en este modelo de placa el tipo "int" utiliza 4 bytes, y por tanto, su valor puede estar dentro de un rango mayor, concretamente entre -2,147,483,648 (-2^{31}) y 2,147,483,647 ($2^{31}-1$) .

El tipo "word": las variables de tipo "word" en la placa Arduino Due ocupan 4 bytes para almacenar su valor. Por tanto, tienen el mismo número de combinaciones numéricas posibles diferentes que las variables "int", pero a diferencia de estas, los valores de una variable "word" no pueden ser negativos. En las placas basadas en microcontroladores de tipo AVR ocurre lo mismo: las variables de tipo "int" y "word" ocupan el mismo espacio de memoria (aunque en este caso, no obstante, solamente son 2 bytes) pero los valores de las segundas no pueden ser negativos. Es fácil ver que el valor que puede tener una variable "word" en todas las placas excepto la Arduino Due es un número entero entre 0 y 65535 ($2^{16}-1$).

El tipo "short": el valor que puede tener una variable de este tipo para todos los modelos de placa (ya sean basadas en microcontroladores de tipo AVR –la mayoría– o de tipo ARM –la Due–) es un número entero entre -32768 (-2^{15}) y 32767 ($2^{15}-1$), gracias a que utilizan 2 bytes (16 bits) de memoria para almacenarse. En este sentido, los tipos "short" e "int" para placas de la familia AVR son equivalentes, pero para la placa Arduino Due el tipo "short" es el único que utiliza 16 bits.

El tipo "long": el valor que puede tener una variable de este tipo para todos los modelos de placa (ya sean basadas en microcontroladores de tipo AVR o de tipo ARM) es un número entero entre -2.147.483.648 y 2.147.483.647 gracias a que utilizan 4 bytes (32 bits) de memoria para almacenarse. En este sentido, los tipos "long" e "int" para placas de la familia ARM son equivalentes.

El tipo "unsigned long": el valor que puede tener una variable de este tipo para todos los modelos de placa (ya sean basadas en microcontroladores de tipo AVR o ARM) es un número entero entre 0 y 4.294.967.295 (2^{32}-1). Al igual que las variables de tipo "long", las de tipo "unsigned long" utilizan 4 bytes (32 bits) para almacenar su valor, y por tanto, tienen el mismo número de combinaciones numéricas posibles diferentes (2^{32}), pero a diferencia de aquellas, los valores de una variable "unsigned long" no pueden ser negativos (tal como ya indica su propio nombre). En este sentido, los tipos "unsigned long" y "word" para placas de la familia ARM son equivalentes.

Breve nota sobre el uso de los sistemas binario y hexadecimal:

Es posible asignar a una variable entera positiva (es decir, de tipo "byte", "word" o "unsigned long") un valor numérico en formato binario o bien hexadecimal, además del conocido formato decimal. Los formatos binario y hexadecimal son maneras diferentes de escribir un número.

En el formato binario los números se representan utilizando solamente combinaciones de los símbolos 0 y 1. Así, por ejemplo, el número 37 (en decimal) se escribe en formato binario así: 100101.

En el formato hexadecimal, los números se representan mediante 16 símbolos (0, 1, 2, 3, 4, 5 ,6 ,7 ,8 ,9 ,A ,B ,C ,D ,E y F), cada uno de los cuales es equivalente a una combinación específica de cuatro valores binarios. En concreto: 0=0000, 1=0001, 2=0010, 3=0011, 4=0100, 5=0101, 6=0110, 7=0111, 8=1000, 9=1001, A=1010, B=1011, C=1100, D=1101, E=1110 y F=1111.

El sistema binario es muy utilizado en la informática y electrónica debido al comportamiento digital que es inherente a los dispositivos tecnológicos, y el sistema hexadecimal no deja de ser una manera de "abreviar" la escritura de números binarios, ya que se utiliza para compactar su representación: por cada cuatro cifras binarias se escribe una cifra hexadecimal.

Las reglas matemáticas para pasar un número escrito en decimal a formato binario, o viceversa, o de decimal a formato hexadecimal, o viceversa, se salen fuera del alcance de este libro. Si se desean conocer, se puede consultar el artículo de la Wikipedia titulado "Sistema binario". De todas formas, estas conversiones de formato se pueden realizar rápidamente mediante la opción adecuada de la aplicación calculadora incluida por defecto en cualquier sistema operativo moderno.

> Si se desea asignar a una variable un valor en formato binario, hay que preceder dicho valor mediante la letra "B" (por ejemplo, así: B10010). Si se desea asignar un valor en formato hexadecimal, hay que preceder dicho valor mediante el prefijo "0x" (por ejemplo, así: 0x54FA). Por lo tanto, una asignación posible podría ser por ejemplo: `unavariable = B1010011;`, donde "1010011" representa el número 83 en decimal (y el 53 en hexadecimal).

En los párrafos anteriores se ha comentado, en referencia a las variables numéricas, que todas ellas tienen un rango de valores válido, y por tanto, asignar un valor fuera de este puede tener consecuencias inesperadas. Esto hay que tenerlo en cuenta para saber elegir correctamente el tipo de variable numérica que necesitemos. En el caso de las variables enteras, lo que uno podría pensar es utilizar siempre las de tipo "unsigned long", las cuales admiten el rango más elevado. Pero esto no es buena idea, porque cada variable de este tipo ocupa cuatro veces más que una variable "byte", y si (por ejemplo) sabemos de antemano que los valores a almacenar no serán mayores de 100, sería una completa pérdida de memoria el utilizar variables "unsigned long". Y la memoria del microcontrolador es uno de los recursos más preciados y escasos para irlo desaprovechando por una mala elección de tipos.

En el supuesto caso de que un valor supere el rango válido (tanto por "encima" como por "debajo"), lo que ocurrirá es que "dará la vuelta", y el valor continuará por el otro extremo. Es decir, si tenemos una variable de tipo "byte" (por ejemplo) cuyo valor es actualmente 255 y se le suma 1, su nuevo valor será entonces 0 (si se le sumara 2, valdría entonces 1, y así). Lo mismo pasa si se supera el límite inferior: si tenemos una variable de tipo "byte" (por ejemplo) cuyo valor actual es 0 y se le resta 1, su nuevo valor será entonces 255 (si se le restara 2, valdría entonces 254, y así). De hecho, esto lo podemos observar si ejecutamos el sketch de ejemplo 4.1 y esperamos lo suficiente: cuando los valores de la variable "mivariable" (de tipo "int", recordemos) lleguen a su límite superior (32767), veremos cómo automáticamente el siguiente valor vuelve a ser el del principio de su rango (concretamente el -32768) para seguir aumentando sin parar otra vez hasta llegar al máximo y volver a caer de nuevo al mínimo, y así. Este fenómeno se llama "overflow".

Existen más tipos de datos admitidos por el lenguaje Arduino además de los ya comentados:

El tipo "float": el valor que puede tener una variable de este tipo es un número decimal. Los valores "float" posibles pueden ir desde el número

$-3,4028235 \cdot 10^{38}$ hasta el número $3,4028235 \cdot 10^{38}$. Debido a su grandísimo rango de valores posibles, los números decimales son usados frecuentemente para aproximar valores analógicos continuos. No obstante, solo tienen 6 o 7 dígitos en total de precisión. Es decir, los valores “float” no son exactos, y pueden producir resultados inesperados, como por ejemplo que, 6.0/3.0 no dé exactamente 2.0.

Otro inconveniente de los valores de tipo “float” es que el cálculo matemático con ellos es mucho más lento que con valores enteros, por lo que debería evitarse el uso de valores “float” en partes de nuestro sketch que necesiten ejecutarse a gran velocidad.

Los números decimales se han de escribir en nuestro sketch utilizando la notación anglosajona (es decir, utilizando el punto decimal en vez de la coma). Si lo deseamos, también se puede utilizar la notación científica (es decir, el número 0,0234 –equivalente a $2,34 \cdot 10^{-2}$ – lo podríamos escribir por ejemplo como 2.34e-2).

El tipo “double”: Es un sinónimo exactamente equivalente del tipo “float”, y por tanto no aporta ningún aumento de precisión respecto a este (a diferencia de lo que pasa en otros lenguajes, donde “double” sí que aporta el doble de precisión). Tanto una variable de tipo “double” como una de tipo “float” ocupan cuatro bytes de memoria.

El tipo “array”: este tipo de datos en realidad no existe como tal. Lo que existen son arrays de variables de tipo “boolean”, arrays de variables de tipo “int”, arrays de variables de tipo “float”, etc. En definitiva: arrays de variables de cualquier tipo de los mencionados hasta ahora. Un array (también llamado “vector”) es una colección de variables de un tipo concreto que tienen todas el mismo y único nombre, pero que pueden distinguirse entre sí por un número a modo de índice. Es decir: en vez de tener diferentes variables –por ejemplo de tipo “char” – cada una independiente de las demás (*varChar1, varChar2,varChar3...*) podemos tener un único array que las agrupe todas bajo un mismo nombre (por ejemplo, *varChar*), y que permita que cada variable pueda manipularse por separado gracias a que dentro del array cada una está identificada mediante un índice numérico, escrito entre corchetes (*varChar[0], varChar[1], varChar[2]...*). Los arrays sirven para ganar claridad y simplicidad en el código, además de facilitar la programación.

Podemos crear –declarar– un array (ya sea en la zona de declaraciones globales o bien dentro de alguna sección concreta), de las siguientes maneras:

`int varInt[6];` Declara un array de 6 elementos (es decir, variables individuales) sin inicializar ninguno.

`int varInt[] = {2,5,6,7};` Declara un array sin especificar el número de elementos. No obstante, se asignan (entre llaves, separados por comas) los valores directamente a los elementos individuales, por lo que el compilador es capaz de deducir el número de elementos total del array (en el ejemplo de la derecha, cuatro).

`int varInt[8] = {2,5,6,7};` Declara un array de 8 elementos e inicializa algunos de ellos (los cuatro primeros), dejando el resto sin inicializar. Lógicamente, si se inicializaran más elementos que lo que permite el tamaño del array (por ejemplo, si se asignan 9 valores a un array de 8 elementos), se produciría un error.

```
char varChar[6] = “hola”;
char varChar[6] = {'h','o','l','a'};
char varChar[] =”hola”;
```

La primera forma declara e inicializa un array de seis elementos de tipo “char”. Como los arrays de tipo “char” son en realidad cadenas de caracteres (es decir: palabras o frases, “strings” en inglés) tienen la particularidad de poder inicializarse tal como se muestra en la primera forma: indicando directamente la palabra o frase escrita entre comillas dobles. Pero también se pueden declarar como un array “estándar”, que es como muestra la segunda forma. Observar en este caso la diferencia de comillas: un carácter individual siempre se especifica entre comillas simples, y el valor de una cadena siempre se especifica entre comillas dobles. También es posible, tal como muestra la tercera forma, declarar una cadena sin necesidad de especificar su tamaño (ya que el compilador lo puede deducir a partir del número de elementos – es decir, caracteres– inicializados).

Hay que tener en cuenta que el primer valor del array tiene el índice 0 y por tanto, el último valor tendrá un índice igual al número de elementos del array menos uno. Cuidado con esto, porque asignar valores más allá del número de elementos declarados del array es un error. En concreto, si por ejemplo tenemos un array de 2 elementos de tipo entero (es decir, declarado así: `int varInt[2];`), para asignar un nuevo valor (por ejemplo, 27) a su primer elemento deberíamos escribir: `varInt[0] = 27;` , y para asignar el segundo valor (por ejemplo, 17), escribiríamos

`varInt[1] = 17;` . Pero si asignáramos además un tercer valor así, `varInt[2] = 53;` , cometeríamos un error porque estaríamos sobrepasando el final previsto del array (y por tanto, utilizando una zona de la memoria no reservada, con resultados imprevisibles).

Por otro lado, además de asignar a un elemento de un array un valor explícito tal como se acaba de comentar en el párrafo anterior, también es posible asignar a un elemento de un array el valor que tenga en ese momento otra variable independiente (preferiblemente del mismo tipo). Por ejemplo, mediante la línea `varInt[4]=x;` estaremos asignando el valor actual de una variable llamada "x" al quinto elemento (¡el índice empieza por 0!) del array llamado "varInt". Y a la inversa también es posible: para asignar el valor que tenga en ese momento el quinto elemento del array "varInt" a la variable independiente "x", simplemente deberemos ejecutar la línea: `x=varInt[4];`

En el caso concreto de los arrays de caracteres (las "cadenas" o "strings"), hay que tener en cuenta una particularidad muy importante: este tipo de arrays siempre deben ser declarados con un número de elementos una unidad mayor que el número máximo de caracteres que preveamos guardar. Es decir, si se va a almacenar la palabra "hola" (de cuatro letras), el array deberá ser declarado como mínimo de 5 elementos. Esto es así porque este último elemento siempre se utiliza para almacenar automáticamente un carácter especial (el carácter "nulo", con código ASCII 0), que sirve para marcar el final de la cadena. Esta marca es necesaria para que el compilador sepa que la cadena ya terminó y no intente seguir leyendo más posiciones. Si no sabemos de antemano cuál es la longitud del texto que se guardará en un array de caracteres, podemos declarar este con un número de elementos lo suficientemente grande como para que haya elementos sin ser asignados, aun sabiendo que así posiblemente estaremos desaprovechando memoria del microcontrolador.

Finalmente, comentar que a menudo es conveniente, cuando se trabaja con grandes cantidades de texto (por ejemplo, en un proyecto con pantallas LCD), utilizar arrays de strings. Para declarar un array de esta clase, se utiliza el tipo de datos especial "char*" (notar el asterisco final). Un ejemplo de declaración e inicialización de este tipo sería: `char* varCadenas[]={"Cad0","Cad1","Cad2","Cad3"};` .

En realidad, el asterisco en la declaración anterior indica que en realidad estamos declarando un array de "punteros", ya que para el lenguaje Arduino, las cadenas son punteros. Los "punteros" son unos elementos del lenguaje Arduino (provenientes del lenguaje C en el que está basado) muy potentes pero a la vez

ciertamente complejos. En este libro no se tratarán, debido a que sus posibles usos son avanzados y pueden confundir al lector que se inicia en la programación: lo único que necesitaremos saber es cómo se declaran los arrays de cadenas y nada más. La buena noticia es que, una vez declarado el array de cadenas, podemos trabajar con él (asignando valores a sus elementos, consultándolos, etc.) como cualquier otro tipo de array sin notar para nada que estamos usando punteros.

Cambio de tipo de datos (numéricos)

No se puede cambiar nunca el tipo de una variable: si esta se declaró de un tipo concreto, seguirá siendo de ese tipo a lo largo de todo el sketch. Pero lo que sí se puede hacer es cambiar "al vuelo" en un momento concreto el tipo del valor que contiene. Esto se llama "casting", y puede ser útil cuando se quiere utilizar ese valor en cálculos que requieran un tipo diferente del original, o también cuando se quiere asignar el valor de una variable de un tipo a otra variable de tipo diferente. Para convertir un valor –del tipo que sea– en otro, podemos utilizar alguna de las siguientes instrucciones:

`char()` : escribiremos entre ( ) el valor –o el nombre de la variable que lo contiene- que queremos convertir en tipo "char"

`byte()` : escribiremos entre ( ) el valor –o el nombre de la variable que lo contiene– que queremos convertir en tipo "byte"

`int()` : escribiremos entre ( ) el valor –o el nombre de la variable que lo contiene– que queremos convertir en tipo "int"

`word()` : escribiremos entre ( ) el valor –o el nombre de la variable que lo contiene– que queremos convertir en tipo "word"

`long()` : escribiremos entre ( ) el valor –o el nombre de la variable que lo contiene– que queremos convertir en tipo "long"

`float()`: escribiremos entre ( ) el valor –o el nombre de la variable que lo contiene– que queremos convertir en tipo "float"

Ejemplo 4.2: Veamos el siguiente código ilustrativo:

```
float variablefloat=3.4;
byte variablebyte=126;
void setup() {
          Serial.begin(9600);
          Serial.println(byte(variablefloat));
          Serial.println(int(variablefloat));
          Serial.println(word(variablefloat));
          Serial.println(long(variablefloat));
```



```
          Serial.println(char(variablebyte));
          Serial.println(float(variablebyte));
}
void loop() {
  //No se ejecuta nada aquí
}
```

Se puede comprobar, una vez ejecutado el sketch anterior, que todas las conversiones del valor float a valores de tipo entero (indiferentemente de si son "byte", "int", "word" o "long") truncan el resultado: pasamos de tener un 3,4 a tener un 3. La diferencia está en los bytes que ocupa ese 3 en la memoria. También podemos comprobar que al convertir un valor numérico a tipo "char", se muestra el carácter correspondiente de la tabla ASCII. Finalmente, la conversión de un valor entero en un valor decimal provoca que podamos trabajar precisamente con decimales a partir de entonces.

<u>Ejemplo 4.3</u>: Una situación concreta donde debemos controlar que los tipos de las variables sean los correctos es en el cálculo matemático. Por ejemplo, si se ejecuta el siguiente código, se puede ver que el resultado obtenido es 2 cuando debería ser 2,5.

```
float resultado;
int numerador=5;
int denominador=2;
void setup() {
           Serial.begin(9600);
           resultado=numerador/denominador;
           Serial.println(resultado);
}
void loop() {}
```

¿Por qué pasa esto? Porque tanto las variables "numerador" como "denominador" son enteras, y por tanto, el resultado siempre será entero, a pesar de que lo guardemos en una variable de tipo "float". Esto es así porque el resultado de un cálculo matemático donde intervienen diferentes tipos enteros siempre es del tipo entero con mayor uso de memoria y con signo –para no perder información– pero nunca es decimal. Solo si interviene en el cálculo algún valor de tipo decimal, entonces el resultado será de tipo decimal (aunque seguirá siendo responsabilidad nuestra guardar ese valor en una variable de tipo decimal para que no haya truncamientos).

Para solucionar el problema del código anterior, debemos realizar un “casting” a “float” de uno de los dos elementos de la división (o de los dos si queremos) para que la operación se realice utilizando esos nuevos tipos, y se obtenga un resultado ya “retipeado” listo para ser asignado a la variable de tipo decimal. Es decir, hemos de sustituir `resultado=numerador/denominador;` por, por ejemplo: `resultado=float(numerador)/denominador;`.

Ejemplo 4.4: Otro problema relacionado con el cambio de tipos de datos lo podemos observar en el siguiente código:

```
int numero=100;
long resultado;
void setup() {
           Serial.begin(9600);
           resultado=numero*1000;
           Serial.println(resultado);
}
void loop() {}
```

Al ejecutar el código anterior veremos por el “Serial monitor” un número que no es el resultado correcto de multiplicar 100 por 1000. ¿Por qué, si la variable “resultado” es de tipo “long” y por tanto puede almacenar un número de esta magnitud? Porque el valor que se le asigna es el resultado de multiplicar el valor de “numero” (de tipo “int”) por el número 1000 (que, si no se especifica explícitamente, es de tipo “int” también ya que hemos de saber que todo número escrito literalmente es siempre de tipo “int”). Es decir, se multiplican dos valores de tipo “int”, y el resultado por tanto sí que sobrepasa el rango aceptado por ese tipo de datos. Que la variable “resultado” sea de tipo “long” no influye para que el número obtenido de la multiplicación de dos números “int” deje de ser incorrecto, porque el cálculo de la multiplicación se realiza antes de asignar el resultado a “resultado”. Es decir, cuando se asigna el valor a “resultado”, este ya ha sufrido el overflow. Para evitar esto, lo más fácil sería forzar a que alguno de los elementos presentes en el cálculo sea del mismo tipo que el de la variable “resultado” porque, tal como ya hemos comentado anteriormente, el tipo de datos del número resultante es el mismo que el tipo de datos del operando con mayor capacidad. Por tanto, si por ejemplo hacemos que la variable “numero” sea “long”, el resultado de la multiplicación será de tipo “long”, con lo que no se producirá “overflow” y se asignará finalmente de forma correcta a la variable “resultado”. Es decir:

```
int numero=100;
long resultado;
```



```
void setup() {
          Serial.begin(9600);
          resultado=long(numero)*1000;
          Serial.println(resultado);
}
void loop() {}
```

Ya sabemos que cuando dentro del código Arduino escribimos directamente números, se asume por defecto que son de tipo “int”. No obstante, si tras su valor literal añadimos la letra “U”, su tipo por defecto será “word”, si añadimos “L”, su tipo será “long” y si se añadimos “UL”, su tipo será “unsigned long”. Es decir, una línea como `resultado=numero*1000L;` hubiera conseguido el mismo efecto que la línea presente en el código anterior.

CONSTANTES

Es posible declarar una variable de tal forma que consigamos que su valor (del tipo que sea) permanezca siempre inalterado. Es decir, que su valor no se pueda modificar nunca porque esté marcado como de “solo lectura”. De hecho, a este tipo de variables ya no se les llama así por motivos obvios, sino “constantes”. Las constantes se pueden utilizar como cualquier variable de su mismo tipo, pero si se intenta cambiar su valor, el compilador lanzará un error.

Para convertir una variable (sea global o local) en constante, lo único que hay que hacer es preceder la declaración de esa variable con la palabra clave `const`. Por ejemplo, para convertir en constante una variable llamada “sensor” de tipo “byte”, simplemente se ha de declarar así: `const byte sensor;` .

Existe otra manera de declarar constantes en el lenguaje Arduino, que es utilizando la directiva especial `#define` (heredada del lenguaje C). No obstante, se recomienda el uso de `const` por su mayor flexibilidad y versatilidad.

PARÁMETROS DE UNA INSTRUCCIÓN

Antes de empezar a aprender y a utilizar las diferentes instrucciones que ofrece el lenguaje Arduino, debemos tener claro un concepto fundamental: los parámetros de una instrucción. Ya se habrá observado que, en el sketch del ejemplo 4.1, al final del nombre de cada instrucción utilizada (*Serial.begin(), Serial.println()*,

etc.) siempre aparecen unos paréntesis. Estos paréntesis pueden estar vacíos pero también pueden incluir en su interior un número/letra/palabra, o dos, o tres, etc. (si hay más de un valor han de ir separados por comas). Cada uno de estos valores es lo que se denomina un "parámetro", y el número de ellos y su tipo (que no tiene por qué ser el mismo para todos) dependerá de cada instrucción en particular.

Los parámetros sirven para modificar el comportamiento de la instrucción en algún aspecto. Es decir: las instrucciones que no tienen parámetros hacen una sola función (su tarea encomendada) y punto: no hay posibilidad de modificación porque siempre harán lo mismo de la misma manera. Cuando una instrucción, en cambio, tiene uno o más parámetros, también hará su función preestablecida, pero la manera concreta vendrá dada por el valor de cada uno de sus parámetros, los cuales modificarán alguna característica concreta de esa acción a realizar.

Por ejemplo: supongamos que tenemos una instrucción llamada *lalala()* que tiene (que "recibe", técnicamente hablando) un parámetro. Supongamos que si el parámetro vale 0 (es decir, si se escribe *lalala(0);*), lo que hará esta instrucción será imprimir por pantalla "Hola, amigo"; si el parámetro vale 1 (es decir, escribiendo *lalala(1);*), lo que hará será imprimir por pantalla "¿Qué tal estás?" y si el parámetro vale 2 (es decir, escribiendo *lalala(2);*) imprimirá "Muy bien", etc. Evidentemente, en las tres posibilidades el comando *lalala()* hace esencialmente lo mismo: imprimir por pantalla (para eso es un comando, sino podríamos estar hablando de tres comandos diferentes), pero dependiendo del valor de su único parámetro, la acción de imprimir por pantalla un mensaje es modificada en cierta manera. Resumiendo: las instrucciones serían como los verbos (ejecutan órdenes), y los parámetros serían como los adverbios (dicen de qué manera se ejecutan esas órdenes).

Observar que en este ejemplo hipotético, se está utilizando un parámetro numérico; no valdría en este caso dar otro tipo de valor (como una letra) porque la instrucción *lalala()* no estaría preparada para recibirlo y daría error. Cada parámetro ha de tener un tipo de datos predefinido.

VALOR DE RETORNO DE UNA INSTRUCCIÓN

Otro concepto fundamental relacionado con las instrucciones del lenguaje Arduino es el de "valor de retorno". Las instrucciones, además de recibir parámetros de entrada (si lo hacen), y aparte de hacer la tarea que tienen que hacer, normalmente también devuelven un valor de salida (o "de retorno"). Un valor de salida es un dato que podemos obtener en nuestro sketch como resultado "tangible"

de la ejecución de la instrucción. El significado de ese valor devuelto dependerá de cada instrucción concreta: algunos son de control (indicando si la instrucción se ha ejecutado bien o mal), otros son resultados numéricos obtenidos tras la ejecución de algún cálculo matemático, etc.

Ya sabemos que cada vez que se llega a una línea en nuestro sketch donde aparece el nombre de la instrucción con sus posibles parámetros, se ejecuta. Lo que no sabíamos es que el valor devuelto de esa ejecución automáticamente "sustituye" dentro del código al nombre de la instrucción y con esta sustitución ya hecha se continúa ejecutando el resto de la línea. Para entenderlo mejor, supongamos que tenemos una instrucción llamada *lalala()* que devuelve un determinado valor. Si queremos usar el valor devuelto podemos:

Asignar ese valor a una variable del mismo tipo, y así poder usarlo posteriormente. Es decir, si esa variable la llamáramos por ejemplo *unavariable*, deberíamos escribir algo parecido a: *unavariable=lalala();*. Fijarse que en este caso, tal como hemos dicho, una vez ejecutada la instrucción, su nombre es "sustituido" por su valor devuelto, y seguidamente este es asignado a *unavariable*.

Utilizar ese valor (que insistimos: "sustituye" al nombre de la instrucción allí donde esté escrito cuando esta se ejecuta) **directamente dentro de otra instrucción**. Por ejemplo, para ver el valor devuelto por *lalala()* podríamos ejecutarla y enviar al "Serial monitor" ese valor, todo de una sola vez, así: `Serial.println(lalala());`

Si no se desea utilizar en ningún momento el valor devuelto, no hace falta hacer nada especial: simplemente ejecutar la instrucción de la forma habitual y listo.

LA COMUNICACIÓN SERIE CON LA PLACA ARDUINO

Ya hemos explicado en capítulos anteriores que el microcontrolador ATmega328P dispone de un receptor/transmisor serie de tipo TTL-UART que permite comunicar la placa Arduino UNO con otros dispositivos (normalmente, nuestro computador), para así poder transferir datos entre ambos. El canal físico de comunicación en estos casos suele ser el cable USB, pero también pueden ser los pines digitales 0 (RX) y 1 (TX) de la placa. Si se usan estos dos pines para comunicar la placa con un dispositivo externo, tendremos que conectar concretamente el pin TX de la placa con el pin RX del dispositivo, el RX de la placa con el TX del dispositivo y

compartir la tierra de la placa con la tierra del dispositivo. Hay que tener en cuenta que si se utilizan estos dos pines para la comunicación serie, no podrán ser usados entonces como entradas/salidas digitales estándar.

Dentro de nuestros sketches podemos hacer uso de este receptor/transmisor TTL-UART para enviar datos al microcontrolador (o recibirlos de él) gracias al elemento del lenguaje Arduino llamado "Serial". En realidad, "Serial" es lo que llamamos un "objeto" del lenguaje. Los objetos son entidades que representan elementos concretos de nuestro sketch. El concepto de objeto es algo abstracto, pero para entenderlo mejor, simplemente supondremos que son "contenedores" que agrupan diferentes instrucciones con alguna relación entre ellas. Por ejemplo, el objeto "Serial" representa por sí mismo una comunicación serie establecida con la placa, y en nuestro sketch podremos hacer uso de un conjunto de instrucciones disponibles dentro de él que sirven para manipular dicha comunicación serie. Si utilizáramos dos objetos de clase "Serial", podríamos manejar entonces dos conexiones serie diferentes, y cada una sería controlada mediante las instrucciones de su objeto respectivo.

Las instrucciones existentes dentro de un objeto (no todas las instrucciones del lenguaje Arduino pertenecen a un objeto) se escriben siguiendo la sintaxis *nombreObjeto.nombreInstruccion();*. Por eso las instrucciones utilizadas en el sketch de ejemplo del principio de este capítulo, al pertenecer al objeto "Serial", tienen nombres como *Serial.begin()* o *Serial.println()*.

A continuación, explicaremos la sintaxis, funcionamiento y utilidad de las instrucciones incluidas en el objeto "Serial". Empezaremos por una ya conocida:

Serial.begin(): abre el canal serie para que pueda empezar la comunicación por él. Por tanto, su ejecución es imprescindible antes de realizar cualquier transmisión por dicho canal. Por eso normalmente se suele escribir dentro de la sección "void setup()". Además, mediante su único parámetro –de tipo "long" y obligatorio– , especifica la velocidad en bits/s a la que se producirá la transferencia serie de los datos. Para la comunicación con un computador, se suele utilizar el valor de 9600, pero se puede especificar cualquier otra velocidad. Lo que sí es importante es que el valor escrito como parámetro coincida con el que se especifique en el desplegable que aparece en el "Serial Monitor" del IDE de Arduino, o si no la comunicación no estará bien sincronizada y se mostrarán símbolos sin sentido. Esta instrucción no tiene valor de retorno.

También existe la instrucción **Serial.end()**, la cual no tiene ningún argumento ni devuelve nada, y que se encarga de cerrar el canal serie; de esta manera, la comunicación serie se deshabilita y los pines RX y TX vuelven a estar disponibles para la entrada/salida general. Para reabrir el canal serie otra vez, se debería usar de nuevo *Serial.begin().*

La otra instrucción que hemos utilizado en los sketches de ejemplo anteriores es *Serial.println(),* que pertenece a un conjunto de instrucciones que permiten enviar datos desde el microcontrolador hacia su exterior. Estudiémoslas en su conjunto.

Instrucciones para enviar datos desde la placa al exterior

Serial.print(): envía a través del canal serie un dato (especificado como parámetro) desde el microcontrolador hacia el exterior. Ese dato puede ser de cualquier tipo: carácter, cadena, número entero, número decimal (por defecto de dos decimales), etc. Si el dato se especifica explícitamente (en vez de a través de una variable), hay que recordar que los caracteres se han de escribir entre comillas simples y las cadenas entre comillas dobles.

En el caso de que el dato a enviar sea entero, se puede especificar un segundo parámetro opcional que puede valer alguna constante predefinida de las siguientes: BIN, HEX o DEC. En el primer caso se enviará la representación binaria del número, en el segundo, la representación hexadecimal, y en el tercero, la representación decimal (la usada por defecto)

En el caso de que el dato a enviar sea decimal, se puede especificar además un segundo parámetro opcional que indique el número de decimales que se desea utilizar (por defecto son dos).

Su valor de retorno es un dato de tipo “byte” que vale el número de bytes enviados. En el caso de cadenas de caracteres, este valor de retorno coincide con el número de caracteres enviados. El uso o no en nuestro sketch de este valor devuelto dependerá de nuestras necesidades.

La transmisión de los datos realizada por *Serial.print()* es asíncrona. Eso significa que nuestro sketch pasará a la siguiente instrucción y seguirá ejecutándose sin esperar a que empiece a realizarse el envío de los datos. Si este comportamiento no es el deseado, se puede añadir justo después de *Serial.print()* la instrucción **Serial.flush()** –que no tiene ningún parámetro ni devuelve ningún valor de retorno–, instrucción que espera hasta que la transmisión de los datos sea completa para continuar la ejecución del sketch.

Serial.println(): hace exactamente lo mismo que *Serial.print(),* pero además, añade automáticamente al final de los datos enviados dos caracteres extra: el de retorno de carro (código ASCII nº 13) y el de nueva línea (código ASCII nº 10). La consecuencia es que al final de la ejecución de *Serial.println()* se efectúa un salto de línea. Tiene los mismos parámetros y los mismos valores de retorno que *Serial.print()*

Se puede simular el comportamiento de *Serial.println()* mediante *Serial.print()* si se añaden manualmente estos caracteres. El carácter retorno de carro se representa con el símbolo '\r', y el de salto de línea con '\n'. Así pues, `Serial.print("hola\r\n");` sería equivalente a `Serial.println("hola");`. Otro carácter ASCII no imprimible útil que podemos representar es el tabulador (mediante '\t').

Llegados a este punto, ya hemos visto el significado de todas las líneas de nuestro primer sketch de este capítulo, y por tanto, deberíamos de ser capaces de entender su comportamiento. Analicémoslo, pues. Si recordamos el código, primero declarábamos una variable global de tipo "int" y la inicializábamos con un valor de 555. Seguidamente, arrancábamos la ejecución del programa abriendo el canal serie (a una velocidad de 9600 bits/s) para que la placa pudiera establecer comunicación con nuestro computador. Y finalmente, ya en la sección "void loop()", primero enviábamos el valor actual de la variable a nuestro computador (que se supone que es el dispositivo conectado vía USB al canal serie abierto). Seguidamente aumentábamos en una unidad ese valor, para justo volver a enviar ese nuevo valor al computador, y volver a aumentar en una unidad su valor, y volverlo a enviar... así infinitamente hasta que la placa dejara de recibir alimentación. Lo que veríamos por el "Serial monitor" sería precisamente esos valores (uno en cada línea diferente porque *Serial.println()* introduce un salto de línea automático) que irían aumentando de uno en uno sin parar. En el momento que se llegara al valor máximo permitido por el tipo de datos de la variable (que en el caso de un "int" es de 32767) se seguiría por el valor mínimo (-32768) y se seguiría aumentando desde allí, en un ciclo sin fin.

Conviene también saber la existencia de la instrucción *Serial.write()*, parecida a *Serial.print()* pero no igual:

Serial.write(): envía a través del canal serie un dato (especificado como parámetro) desde el microcontrolador hacia el exterior. Pero a diferencia de *Serial.print()*, el dato a enviar solo puede ocupar un byte. Por lo tanto, ha de ser básicamente de tipo "char" o "byte". En realidad, también es capaz de

transmitir cadenas de caracteres porque las trata como una mera secuencia de bytes independientes uno tras otro. En cambio, otros tipos de datos que ocupen más de un byte indisolublemente (como los "int", "word", "float"...) no serán enviados correctamente. Su valor de retorno es, al igual que en *Serial.print(),* un dato de tipo "byte" que vale el número de bytes enviados.

La gracia de *Serial.write()* está en que el dato es enviado siempre directamente sin interpretar. Es decir, se envía como un byte (o una serie de bytes) tal cual sin ninguna transformación de formato. Esto no pasa con *Serial.print()*, en la cual se puede jugar con los formatos binario, hexadecimal, etc. Por tanto, esta instrucción está pensada para la transferencia directa de datos con otro dispositivo sin una previsualización por parte nuestra.

No obstante, si observamos en el "Serial monitor" los datos enviados por *Serial.write()*, veremos que se muestran (tanto si son de tipo "char" como de tipo "byte") en forma de sus correspondientes caracteres ASCII. Esto es así porque es el propio "Serial monitor" el que realiza en tiempo real esta "traducción" ASCII. Gracias a este comportamiento, podemos ver en todo momento el carácter asociado al byte enviado, que es lo que normalmente nos interesa.

Ejemplo 4.5: El siguiente código ilustra mejor el comportamiento recién descrito:

```
char cadena[]="hola";
byte bytesDevueltos;
void setup() {
        Serial.begin(9600);
        bytesDevueltos=Serial.write(cadena);
        Serial.println(bytesDevueltos);
}
void loop() {}
```

Si ejecutamos el código anterior y observamos el "Serial monitor", veremos que aparece el valor "hola4". El 4 final muestra el valor de la variable "bytesDevueltos", que guarda lo devuelto por `Serial.write(cadena);`, confirmando así que se han enviado por el canal serie 4 bytes, los correspondientes precisamente a la cadena "hola".

Existe otra manera de utilizar *Serial.write(),* que es para enviar de golpe un array de datos de tipo "byte". En ese caso, esta instrucción tiene dos parámetros: el nombre de ese array y el número de los elementos (empezando siempre por el

primero) que se quieren enviar. Este último valor no tiene por qué coincidir con el número total de elementos del array.

Ejemplo 4.6: El siguiente código ilustra mejor el comportamiento recién descrito:

```
//El array ha de ser de tipo "byte" (o "char")
byte arraybytes[ ]={65,66,67,68};
void setup()  {
        Serial.begin(9600);
        //Se envían solo los dos primeros elementos de ese array
        Serial.write(arraybytes,2);
}
void loop() {}
```

Sea como sea, la transmisión del byte es asíncrona. Eso significa que nuestro sketch pasará a la siguiente instrucción para seguir ejecutándose sin esperar a que empiece a realizarse el envío de ese/os byte/s. Si este comportamiento no es el deseado, se puede añadir justo después de *Serial.write()* la instrucción **Serial.flush()** –que no tiene ningún parámetro y no devuelve ningún valor de retorno–, la cual esperará hasta que la transmisión de los datos sea completa para continuar la ejecución del sketch.

Instrucciones para recibir datos desde el exterior

Hasta ahora hemos visto instrucciones que permiten al microcontrolador enviar datos a su entorno. Pero ¿cómo se hace a la inversa? Es decir: ¿cómo se pueden enviar datos al microcontrolador (para que este los recoja y los procese) que provengan de su entorno, como por ejemplo nuestro computador?

Desde el "Serial monitor" enviar datos a la placa es muy sencillo: no hay más que escribir lo que queramos en la caja de texto allí mostrada y pulsar el botón "Send". No obstante, si el sketch que se está ejecutando en la placa no está preparado para recibir y procesar estos datos, esa transmisión no llegará a ningún sitio. Por tanto, necesitamos recibir convenientemente en nuestros sketches los datos que lleguen a la placa vía comunicación serie. Para ello, disponemos de dos instrucciones básicas: *Serial.available()* y *Serial.read().*

Serial.available(): devuelve el número de bytes –caracteres– disponibles para ser leídos que provienen del exterior a través del canal serie (vía USB o vía pines TX/RX). Estos bytes ya han llegado al microcontrolador y permanecen almacenados temporalmente en una pequeña memoria de 64 bytes que

tiene el chip TTL-UART –llamada "buffer"– hasta que sean procesados mediante la instrucción *Serial.read().* Si no hay bytes para leer, esta instrucción devolverá 0. No tiene parámetros.

Serial.read(): devuelve el primer byte aún no leído de los que estén almacenados en el buffer de entrada del chip TTL-UART. Al hacerlo, lo elimina de ese buffer. Para devolver (leer) el siguiente byte, se ha de volver a ejecutar *Serial.read().* Y hacer así hasta que se hayan leído todos. Cuando no haya más bytes disponibles, *Serial.read()* devolverá -1. No tiene parámetros.

Ejemplo 4.7: Veamos un código básico de estas nuevas instrucciones:

```
byte byteRecibido = 0;
void setup() {
        Serial.begin(9600);
}
void loop() {
        if (Serial.available() > 0) {
                byteRecibido = Serial.read();
                Serial.write("Byte recibido: ");
                Serial.write(byteRecibido);
        }
}
```

En el código anterior hemos introducido un elemento del lenguaje que todavía no hemos visto: el condicional "if". Lo explicaremos en profundidad en su apartado correspondiente de este capítulo, pero baste por ahora saber que un "if" mira si la condición escrita entre paréntesis es cierta o no: si lo es, se ejecutan las instrucciones escritas dentro de sus llaves, y si no, no. La condición vemos que es *Serial.available() > 0* , así que lo que se está comprobando es si existen o no datos almacenados en el buffer de entrada del chip TTL-UART. Si esta condición es cierta, se ejecuta el interior del "if", que lo que hace básicamente es leer el primer byte disponible que haya en el buffer (eliminándolo de allí) y enviarlo al "Serial monitor". Como todo esto ocurre dentro de la sección "void loop()", seguidamente regresaremos otra vez a su principio para volver a comprobar si aún existen datos almacenados en el buffer de entrada. Si sigue siendo así, se leerá el siguiente byte disponible y se volverá a enviar al "Serial monitor". Y se comprobará otra vez si sigue habiendo datos en el buffer, en cuyo caso se leerá el siguiente byte. Y así hasta que ya se hayan leído todos los bytes disponibles. En ese momento, la condición del "if" pasará a ser falsa (porque *Serial.available()* devolverá 0) y por tanto "void loop()" no ejecutará nada. Pero como a cada repetición de "void loop()" se seguirá

comprobando si hay o no datos en el buffer, en el momento que vuelva a ver, la condición del "if" volverá a ser cierta y por tanto otra vez se empezarán a leer los bytes disponibles allí, uno a uno.

Un detalle muy importante del código anterior es que, tal como se puede observar, el valor devuelto por *Serial.read()* se guarda en una variable de tipo "byte". Esto implica que ese valor se almacena en formato numérico. Es decir: si *Serial.read()* recibe por ejemplo el valor "a", lo que se guarda en una variable de tipo "byte" es su valor numérico correspondiente en la tabla ASCII (en este caso, 97); igualmente, si *Serial.read()* recibe por ejemplo el valor "1", en realidad lo que se guarda en una variable de tipo "byte" es el valor numérico 49, y así. Y esto es independiente de cómo se muestren estos datos por el "Serial monitor". Sin embargo, si el tipo de la variable utilizado para guardar el valor devuelto por *Serial.read()* es "char" (recordemos que es el otro tipo de datos que ocupa 1 byte en memoria), lo que ocurre es que "a" será leído como el carácter "a", "1" como el carácter "1", etc. La consecuencia de todo ello es que debemos pensar previamente qué utilidad le vamos a dar en nuestro sketch al valor devuelto (¿número o carácter?) para entonces decidir el tipo de datos de la variable que lo guardará.

Existen otras instrucciones además de *Serial.read()* que leen datos del buffer de entrada del chip TTL-UART de formas más específicas, las cuales nos pueden venir bien en determinadas circunstancias:

Serial.peek(): devuelve el primer byte aún no leído de los que estén almacenados en el buffer de entrada. No obstante, a diferencia de *Serial.read(),* ese byte leído no se borra del buffer, con lo que las próximas veces que se ejecute *Serial.peek()* –o una vez *Serial.read()*– se volverá a leer el mismo byte. Si no hay bytes disponibles para leer, *Serial.peek()* devolverá -1. Esta instrucción no tiene parámetros.

Serial.find(): lee datos del buffer de entrada (eliminándolos de allí) hasta que se encuentre la cadena de caracteres (o un carácter individual) especificada como parámetro, o bien se hayan leído todos los datos actualmente en el buffer. La instrucción devuelve "true" si se encuentra la cadena o "false" si no.

<u>Ejemplo 4.8</u>: El código siguiente utiliza *Serial.find()* para leer continuamente los datos presentes en el buffer de entrada en busca de la palabra "hola". Se puede probar su comportamiento escribiendo en la caja de texto del "Serial monitor" las cadenas que deseemos enviar a la placa. Si la cadena "hola" se encuentra, se mostrará por el "Serial monitor" la palabra "Encontrado".


```
boolean encontrado;
void setup(){
         Serial.begin(9600);
}
void loop(){
         encontrado=Serial.find("hola");
         if (encontrado == true){
                 Serial.println("Encontrado");
        }
}
```

Nota: aquí volvemos a ver un ejemplo de “if”: básicamente lo que comprueba es que el valor devuelto por *Serial.find()* sea verdadero para así poder imprimir “Encontrado”. Es importante darse cuenta de que se han de escribir los dos iguales en la condición del “if” (ya hablaremos de ello en el apartado correspondiente).

Serial.findUntil(): lee datos del buffer de entrada (eliminándolos de allí) hasta que se encuentre la cadena de caracteres (o un carácter individual) especificada como primer parámetro, o bien se llegue a una marca de final de búsqueda (la cual es la cadena –o carácter– especificada como segundo parámetro). La instrucción devuelve “true” si se encuentra la cadena a buscar antes que la marca de final de búsqueda o “false” si no.

Serial.readBytes(): lee del buffer de entrada (eliminándolos de allí) la cantidad de bytes especificada como segundo parámetro (o bien, si no llegan suficientes bytes, hasta que se haya superado el tiempo especificado por *Serial.setTimeout()*) . En cualquier caso, los bytes leídos se almacenan en un array –de tipo “char[]”– especificado como primer parámetro. Esta instrucción devuelve el número de bytes leídos del buffer (por lo que un valor 0 significa que no se encontraron datos válidos).

Ejemplo 4.9: El código siguiente (donde se ve el uso de *Serial.readBytes()*) obtendrá lo que haya en el buffer de entrada de 20 en 20 bytes, y los almacenará en el array “miarray”, mostrando seguidamente la cantidad de bytes obtenidos y sus valores en forma de cadena. Si en el buffer hay más de 20 bytes, a la siguiente repetición de “void loop()” se volverá a ejecutar *Serial.readBytes()*, con lo que leerá los siguientes 20 bytes del buffer y se sobrescribirán los valores que había en el array por los nuevos. Se puede utilizar el botón “Send” del “Serial monitor” para enviar caracteres a la placa y observar el resultado.

```
char miarray[30];
byte bytesleidos;
void setup(){
        Serial.begin(9600);
}
void loop(){
        bytesleidos=Serial.readBytes(miarray,20);
        Serial.println(bytesleidos);
        Serial.println(miarray);
}
```

Serial.readBytesUntil(): lee del buffer de entrada (eliminándolos de allí) la cantidad de bytes especificada como tercer parámetro, o bien, si se encuentra antes una cadena de caracteres –o carácter individual– especificada como primer parámetro que hace de marca de final, o bien, si no llegan suficientes bytes ni se encuentra la marca de final, hasta que se haya superado el tiempo especificado por *Serial.setTimeout()*. En cualquier caso, los bytes leídos se almacenarán en un array –de tipo "char[]"– especificado como segundo parámetro. Esta instrucción devuelve el número de bytes leídos del buffer (por lo que un valor 0 significa que no se encontraron datos válidos).

Serial.setTimeout(): tiene un parámetro (de tipo "long") que sirve para establecer el número de milisegundos máximo que las instrucciones *Serial.readBytesUntil()* y *Serial.readBytes()* esperarán a la llegada de datos al búfer de entrada serie. Si alguna de estas dos instrucciones no recibe ningún dato y se supera ese tiempo, el sketch continuará su ejecución en la línea siguiente. El tiempo de espera por defecto es de 1000 milisegundos. Esta instrucción se suele escribir en "void setup ()". No tiene valor de retorno.

Serial.parseFloat(): lee del buffer de entrada (eliminándolos de allí) todos los datos hasta que se encuentre con un número decimal. Su valor de retorno –de tipo "long"– será entonces ese número decimal encontrado. Cuando detecte el primer carácter posterior no válido, dejará de leer (y por tanto, no seguirá eliminando datos del buffer). Esta instrucción no tiene parámetros.

Ejemplo 4.10: El siguiente código (junto con la ayuda del botón "Send" del "Serial monitor") nos permite probar el uso de *Serial.parseFloat()*:


```
float numero;
void setup(){
        Serial.begin(9600);
}
void loop(){
/* Vacía el buffer hasta reconocer
algún número decimal o vaciarlo del todo */
        numero=Serial.parseFloat();
/*Imprime el número decimal detectado,
y si no se ha encontrado ninguno, imprime 0.00 */
        Serial.println(numero);
/*Lee un byte más y lo imprime. Si se hubiera detectado un número
decimal, ese byte sería el carácter que está justo después de él. Si
el buffer está vacío porque Serial.parseFloat() no encontró ningún
número decimal, entonces devuelve -1 */
        Serial.println(Serial.read());
}
```

Serial.parseInt(): lee del buffer de entrada (eliminándolos de allí) todos los datos hasta que se encuentre con un número entero. Su valor de retorno –de tipo "long"– será entonces ese número entero encontrado. Cuando detecte el primer carácter posterior no válido, dejará de leer (y por tanto, no seguirá eliminando datos del buffer). Esta instrucción no tiene parámetros.

Los objetos serie de otras placas Arduino

Hasta ahora hemos supuesto el uso de la placa Arduino UNO, la cual tiene solo un chip TTL-UART y permite por tanto un único objeto serie, llamado "Serial". No obstante, otros modelos de placa Arduino disponen de más objetos de este tipo, y por tanto, de una mayor versatilidad en el uso de la comunicación serie.

Por ejemplo, la placa Arduino Mega dispone de cuatro chips TTL-UART. Esto significa que podemos utilizar hasta cuatro objetos serie, llamados "Serial", "Serial1", "Serial2" y "Serial3". El primero sigue estando asociado a los pines 0 y 1, el objeto "Serial1" está asociado a los pines 18 (TX) y 19 (RX), el "Serial2" a los pines 16 (TX) y 17 (RX) y el "Serial3" a los pines 14 (TX) y 15 (RX). Cada uno de estos objetos se puede abrir independientemente (escribiendo `Serial.begin(9600);` `Serial1.begin(9600);` `Serial2.begin(9600);` o `Serial3.begin(9600);` respectivamente), y pueden enviar y recibir datos también independientemente. No obstante, el único objeto asociado también a la conexión USB es "Serial" (porque es el único conectado al chip conversor ATmega16U2).

La placa Arduino Leonardo también dispone, aparte del objeto “Serial”, del objeto “Serial1”. En este caso es para separar las dos posibles vías de comunicación serie que puede manejar el chip TTL-UART incorporado dentro del ATmega32U4: el objeto “Serial1” se reserva para la transmisión de información a través de los pines 0 (RX) y 1 (TX), y el objeto “Serial” se reserva para esa misma transmisión serie pero realizada a través de la comunicación USB-ACM (que a su vez es diferente de la comunicación USB usada para las simulaciones de teclado y ratón).

Hay que tener en cuenta también, que, a diferencia de lo que ocurre con el modelo UNO, la ejecución de un sketch en la placa Leonardo no se reinicia cuando se abre el “Serial monitor”, por lo que no se podrán ver los datos que ya han sido enviados por la placa previamente al computador (como por ejemplo, los enviados dentro de la función “setup()” con *Serial.print()* o similares). Para sortear este inconveniente, se puede escribir la siguiente línea justo después de *Serial.begin()*: `while(!Serial){;}` Esto hará que mientras no se abra la comunicación serie (es decir, mientras no se abra el “Serial monitor”), el sketch no haga nada y se mantenga en espera “latente”. Cuando estudiemos el bucle “while” al final de este capítulo se comprenderá mejor su significado.

La placa Arduino Due, por su parte, dispone de tres puertos serie adicionales, (además del puerto “Serial” ubicado en los pines estándares 0 y 1). Estos puertos extra pueden ser utilizados en nuestros sketches mediante los objetos “Serial1” (correspondiente a los pines 18 –TX– y 19 –RX–), “Serial2” (correspondiente a los pines 16 –TX– y 17 –RX–) y “Serial3” (correspondiente a los pines 14 –TX– y 15 –RX–). De todos ellos, es el objeto “Serial” el que está comunicado con el conector USB mini-B de la placa (a través del chip ATmega16U2), por lo que ese será el objeto que deberemos utilizar en nuestros sketches para enviar y recibir datos serie a través del cable USB. Podríamos enviar y recibir datos serie directamente al chip SAM3X a través del cable USB si conectáramos este al zócalo USB mini-A, pero en este caso el objeto a utilizar en nuestros sketches debería de ser otro, el llamado “SerialUSB”.

En todo caso, hay que tener presente que en la Arduino Due todos los puertos trabajan a 3,3 V (en vez de 5 V como el resto de placas).

INSTRUCCIONES DE GESTIÓN DEL TIEMPO

Estas instrucciones no pertenecen a ningún objeto, así que se escriben directamente:

millis(): devuelve el número de milisegundos (ms) desde que la placa Arduino empezó a ejecutar el sketch actual. Este número se reseteará a cero aproximadamente después de 50 días (cuando su valor supere el máximo permitido por su tipo, que es "unsigned long"). No tiene parámetros.

micros(): devuelve el número de microsegundos (µs) desde que la placa Arduino empezó a ejecutar el sketch actual. Este número –de tipo "unsigned long"– se reseteará a cero aproximadamente después de 70 minutos. Esta instrucción tiene una resolución de 4 µs (es decir, que el valor retornado es siempre un múltiplo de cuatro). Recordar que 1000 µs es un milisegundo y por tanto, 1000000 µs es un segundo. No tiene parámetros.

delay(): pausa el sketch durante la cantidad de milisegundos especificados como parámetro –de tipo "unsigned long"–. No tiene valor de retorno.

delayMicroseconds(): pausa el sketch durante la cantidad de microsegundos especificados como parámetro –de tipo "unsigned long"– . Actualmente el máximo valor que se puede utilizar con precisión es de 16383. Para esperas mayores que esta, se recomienda usar la instrucción *delay()*. El mínimo valor que se puede utilizar con precisión es de 3 µs. No tiene valor de retorno.

Ejemplo 4.11: Un código sencillo de alguna de las instrucciones anteriores es este.

```
unsigned long time;
void setup(){
          Serial.begin(9600);
}
void loop(){
         time = micros();
         Serial.println(time);
         delay(1000);
}
```

Si se ejecuta el código anterior se puede ver por el "Serial monitor" cómo va aumentando el tiempo que pasa desde que se puso en marcha el sketch. El valor observado va aumentando aproximadamente un segundo cada vez.

Ejemplo 4.12: Otro código ilustrativo es el siguiente.

```
unsigned long inicio, fin, transcurrido;
void setup(){
	Serial.begin(9600);
}
void loop(){
	inicio=millis();
	delay(1000);
	fin=millis();
	transcurrido=fin-inicio;
	Serial.print(transcurrido);
	delay(500);
}
```

En el código anterior se puede ver una manera de contar el tiempo transcurrido entre dos momentos determinados. El procedimiento es guardar en una variable el valor devuelto por *millis()* en el momento inicial, y guardar en otra variable diferente el valor devuelto por *millis()* en el momento final, para seguidamente restar uno del otro y averiguar así el lapso de tiempo transcurrido (que en el ejemplo debería de ser de aproximadamente un segundo). Este cálculo en el código anterior se realiza cada medio segundo.

INSTRUCCIONES MATEMÁTICAS, TRIGONOMÉTRICAS Y DE PSEUDOALEATORIEDAD

El lenguaje Arduino dispone de una serie de instrucciones matemáticas y de pseudoaleatoriedad que nos pueden venir bien en nuestros proyectos. Son estas:

abs(): devuelve el valor absoluto de un número pasado por parámetro (el cual puede ser tanto entero como decimal). Es decir, si ese número es positivo (o 0), lo devuelve sin alterar su valor; si es negativo, lo devuelve "convertido en positivo". Por ejemplo, 3 es el valor absoluto tanto de 3 como de -3.

min(): devuelve el mínimo de dos números pasados por parámetros (los cuales pueden ser tanto enteros como decimales).

max(): devuelve el máximo de dos números pasados por parámetros (los cuales pueden ser tanto enteros como decimales).

Muchas veces se utilizan las funciones *min()* y *max()* para restringir el valor mínimo o máximo que puede tener una variable cuyo valor proviene de un sensor.

Por ejemplo, si tenemos una variable llamada "sensVal" que obtiene su valor de un sensor, la línea `sensVal = min(sensVal, 100);` se asegura que dicha variable no va a tener nunca un valor más pequeño de 100, a pesar de que ese sensor reciba en un momento dado valores más pequeños. En este sentido, nos puede ser útil otra instrucción más específica llamada *constrain(),* la cual sirve para contener un valor determinado entre dos extremos mínimo y máximo:

constrain(): recalcula el valor pasado como primer parámetro (llamémosle "x") dependiendo de si está dentro o fuera del rango delimitado por los valores pasados como segundo y tercer parámetro (llamémoslos "a" y "b" respectivamente, donde "a" siempre ha de ser menor que "b"). Los tres parámetros pueden ser tanto enteros como decimales. En otras palabras:

Si "x" está entre "a" y "b", *constrain()* devolverá "x" sin modificar.
Si "x" es menor que "a", *constrain()* devolverá "a"
Si "x" es mayor que "b", *constrain()* devolverá "b"

Una instrucción algo más compleja que las anteriores (pero más versátil) es la instrucción *map().* Esta instrucción es utilizada en multitud de proyectos para adecuar las señales de entrada obtenidas por diferentes sensores a un rango numérico óptimo para trabajar. Veremos varios ejemplos de su uso en posteriores apartados.

map(): modifica un valor –especificado como primer parámetro– el cual inicialmente está dentro de un rango (delimitado con su mínimo –segundo parámetro– y su máximo –tercer parámetro–) para que esté dentro de otro rango (con otro mínimo –cuarto parámetro– y otro máximo –quinto parámetro–) de forma que la transformación del valor sea lo más proporcional posible. Esto es lo que se llama "mapear" un valor: los mínimos y los máximos del rango cambian y por tanto los valores intermedios se adecúan a ese cambio. Todos los parámetros son de tipo "long", por lo que se admiten también números enteros negativos, pero no números decimales: si aparece alguno en los cálculos internos de la instrucción, éste será truncado. El valor devuelto por esta instrucción es precisamente el valor mapeado.

Ejemplo 4.13: El siguiente código muestra el uso de la instrucción *map().*

```
void setup(){
	Serial.begin(9600);
```

```
       Serial.println(map(0,0,100,200,400));
       Serial.println(map(25,0,100,200,400));
       Serial.println(map(50,0,100,200,400));
       Serial.println(map(75,0,100,200,400));
       Serial.println(map(100,0,100,200,400));
      /*El valor puede estar fuera de los rangos,
      pero igualmente se mapea de la forma conveniente */
       Serial.println(map(500,0,100,200,400));
}
void loop(){}
```

Si se observa la salida en el “Serial monitor”, se puede comprobar cómo el valor mínimo del rango inicial (0) se mapea al valor mínimo del rango final (200), que el valor 25 (una cuarta parte del rango total inicial) se mapea al valor 250 (una cuarta parte del rango total final), que el valor 50 (la mitad del rango total inicial) se mapea al valor 300 (la mitad del rango total final), que el valor 75 (tres cuartas partes del rango total inicial) se mapea al valor 350 (tres cuartas partes del rango total final), que el valor máximo del rango inicial (100) se mapea al valor máximo del rango final (400) y que un valor fuera del rango inicial (500) se mapea proporcionalmente a otro valor fuera del rango final (1200).

Observar en este último caso que los valores mapeados no se restringen dentro del nuevo rango porque a veces puede ser útil trabajar con valores fuera de rango. Si se desea acotarlos, se debe usar la instrucción *constrain()* o antes o después de *map()*.

Ver también que los mínimos de los dos rangos (el actual y el modificado) pueden tener en realidad un valor mayor que los máximos. De esta forma, se pueden revertir rangos de números. Por ejemplo, *map(x,1, 50, 50 ,1);* invertiría los valores originales: el 1 pasaría a ser el 50, el 2 el 49… y el 50 pasaría a ser el 1, el 49 el 2…

Para los curiosos: matemáticamente, la instrucción *map()*realiza este cálculo: *xmapeado =(x– minactual)*(maxfinal – minfinal) / (maxactual – minactual) + minfinal* donde ”x” es el valor a mapear,”minactual” es el valor mínimo del rango actual, “maxactual” es el valor máximo del rango actual, “minfinal” es el valor mínimo del rango final y “maxfinal” es el valor máximo del rango final.

El lenguaje Arduino también dispone de instrucciones relacionadas con potencias:

pow(): devuelve el valor resultante de elevar el número pasado como primer parámetro (la "base") al número pasado como segundo parámetro (el "exponente", el cual puede ser incluso una fracción). Por ejemplo, si ejecutamos `resultado = pow (2,5);` la variable "resultado" valdrá 32 (2^5). Ambos parámetros son de tipo "float", y el valor devuelto es de tipo "double".

Si se quiere calcular el cuadrado de un número (es decir, el resultado de multiplicar ese número por sí mismo, además de utilizar la instrucción *pow()* poniendo como exponente el número 2, se puede utilizar una instrucción específica que es **sq()**, cuyo único parámetro es el número (de cualquier tipo) que se desea elevar al cuadrado y cuyo resultado se devuelve en forma de número de tipo "double".

sqrt(): devuelve la raíz cuadrada del número pasado como parámetro (que puede ser tanto entero como decimal). El valor devuelto es de tipo "double".

También podemos utilizar instrucciones trigonométricas. Va más allá de los objetivos de este libro explicar la utilidad y significado matemático de estas funciones; si el lector desea profundizar en sus conocimientos de trigonometría, recomiendo consultar el artículo de la Wikipedia titulado "Función trigonométrica".

sin(): devuelve el seno de un ángulo especificado como parámetro en radianes. Este parámetro es de tipo "float". Su retorno puede ser un valor entre -1 y 1 y es de tipo "double".

cos(): devuelve el coseno de un ángulo especificado como parámetro en radianes. Este parámetro es de tipo "float". Su retorno puede ser un valor entre -1 y 1 y es de tipo "double".

tan(): devuelve la tangente de un ángulo especificado como parámetro en radianes. Este parámetro es de tipo "float". Su retorno puede ser un valor entre -∞ y ∞ y es de tipo "double".

También podemos utilizar números pseudoaleatorios en nuestros sketches Arduino. Un número pseudoaleatorio no es estrictamente un número aleatorio según la definición matemática rigurosa, pero para nuestros propósitos el nivel de aleatoriedad que alcanzan las siguientes funciones será más que suficiente:

randomSeed(): inicializa el generador de números pseudoaleatorios. Se suele ejecutar en la sección "setup()" para poder utilizar a partir de entonces números pseudoaleatorios en nuestro sketch. Esta instrucción tiene un parámetro de tipo "int" o "long" llamado "semilla" que indica el valor a partir del cual empezará la secuencia de números. Semillas iguales generan secuencias iguales, así que interesará en múltiples ejecuciones de *randomSeed()* utilizar valores diferentes de semilla para aumentar la aleatoriedad. También nos puede interesar a veces lo contrario: fijar la semilla para que la secuencia de números aleatorios se repita exactamente. No tiene ningún valor de retorno.

random(): una vez inicializado el generador de números pseudoaleatorios con *randomSeed()*, esta instrucción retorna un número pseudoaleatorio de tipo "long" comprendido entre un valor mínimo (especificado como primer parámetro –opcional–) y un valor máximo (especificado como segundo parámetro) menos uno. Si no se especifica el primer parámetro, el valor mínimo por defecto es 0. El tipo de ambos parámetros puede ser cualquiera mientras sea entero.

Ejemplo 4.14: Un ejemplo de uso de las instrucciones pseudoaleatorias sería el siguiente sketch. Ahí se puede observar que se tiene una semilla fija y que se generarán números pseudoaleatorios entre 1 y 9. Observar que, al tener la semilla fija, si se abre el "Serial monitor" en diferentes ocasiones (suponiendo que usamos una placa UNO), en todas ellas la secuencia de números será exactamente la misma:

```
void setup(){
      Serial.begin(9600);
      randomSeed(100);
}
void loop(){
      Serial.println(random(1,10));
      delay(1000);
}
```

Además de las instrucciones matemáticas anteriores, el lenguaje Arduino dispone de varios operadores aritméticos, algunos de los cuales ya han ido apareciendo en algunos códigos de ejemplo. Estos operadores funcionan tanto para números enteros como decimales y son los siguientes:

Operadores aritméticos	
+	Operador suma
-	Operador resta
*	Operador multiplicación
/	Operador división
%	Operador módulo

El operador módulo sirve para obtener el resto de una división. Por ejemplo: 27%5=2 . Es el único operador que no funciona con "floats".

Un comentario final: posiblemente, a priori las funciones matemáticas del lenguaje Arduino puede parecer escasas comparadas con las de otros lenguajes de programación. Pero nada más lejos de la realidad: precisamente porque el lenguaje Arduino no deja de ser un "maquillaje" del lenguaje C/C++, en realidad tenemos a nuestra disposición la mayoría de funciones matemáticas (y de hecho, prácticamente cualquier tipo de función) que ofrece el lenguaje C/C++. Concretamente, podemos utilizar todas las funciones listadas en la referencia online de "avr-libc" (http://www.nongnu.org/avr-libc/user-manual) . Así pues, si necesitamos calcular el exponencial de un número decimal "x", podemos usar `exp(x);`. Si queremos calcular su logaritmo natural, podemos usar `log(x);`. Si queremos calcular su logaritmo en base 10, podemos usar `log10(x);`. Si queremos calcular el módulo de una división de dos números decimales, podemos usar `fmod(x,y);` etc.

INSTRUCCIONES DE GESTIÓN DE CADENAS

El lenguaje Arduino incluye un completo conjunto de instrucciones de manipulación y tratamiento de cadenas. Gracias a ellas se pueden buscar cadenas dentro de otras, sustituir una cadena por otra, unir ("concatenar") cadenas , conocer su longitud, etc.

No obstante, todas estas instrucciones pertenecen a un objeto específico del lenguaje llamado "String", por lo que no se pueden utilizar con un simple array de caracteres. Así pues, para poder empezar a utilizar todas estas funciones con una o más cadenas, lo más común es declarar esas cadenas de tipo "String" (notar la S mayúscula) en vez de como un array de caracteres. Es decir, escribir una declaración como esta: `String unacadena="hola qué tal";`. También se puede declarar

como String un carácter individual, así: `String uncaracter='a';` O inicializar un nuevo objeto String con el valor de otro ya existente: `String objetostringnuevo = objetostringyaexistente;`.

Si lo que queremos es convertir valores que inicialmente eran numéricos enteros (“byte”, “int”, ”word”, ”long”, ”unsigned long”) en objetos String (concretamente, en su representación ASCII), deberemos utilizar la instrucción especial *String().* Así, para declarar una variable de tipo String cuyo valor era inicialmente un número se debe escribir: `String unacadena=String(13);` : esto hará que tengamos un objeto String llamado “unacadena” con el valor “13”. La instrucción *String()* es muy flexible, y permite que se escriban en su interior no solamente números exactos (como el 13 del ejemplo anterior) sino también valores extraídos de variables de tipo entero o de instrucciones que devuelvan ese tipo de valores (por ejemplo: `String unacadena=String(unavariableentera);`.

En estos casos donde hay una conversión de valores numéricos a valores String, se puede escribir opcionalmente dentro de la instrucción *String()* un segundo parámetro que puede valer las constantes predefinidas BIN o HEX , indicando si el valor de tipo String estará en formato binario o hexadecimal, respectivamente. Es decir, si por ejemplo tenemos la siguiente declaración: `String unacadena=String(45, BIN);`, el valor de “unacadena” será “00101101”, y si tenemos `String unacadena=String(45, HEX);`, será “2D”.

Otra característica más que podemos hacer es concatenar el valor de un objeto String con otros valores (escritos explícitamente, o guardados en alguna variable, o devueltos por alguna instrucción) que pueden ser de tipo String pero también arrays de caracteres, o caracteres individuales, o incluso valores numéricos representados por sus cifras ASCII. Para ello, debemos utilizar el signo “+”. Es decir, si tenemos por ejemplo las declaraciones `String unacadena=”hola”; char otracadena[]=”adiós”; char uncaracter='o';` y `byte unnumero=123;` para asignar a “unacadena” el valor “holaadioso123” podemos hacer simplemente `unacadena= unacadena + otracadena + uncaracter + unnumero;` (sobrescribiendo el valor que tenía anteriormente, lógicamente). También se lo podríamos asignar a un tercer objeto String diferente (llamémoslo “otramas”): `otramas = unacadena + otracadena + uncaracter + unnumero;`. La concatenación de cadenas es muy útil cuando se quiere mostrar una combinación de valores junto con su descripción en una sola cadena (en una pantalla LCD, en una conexión Ethernet o serie, etc.).

Podemos hacer incluso dos cosas a la vez: declarar un objeto String e inicializarlo en ese mismo momento con la unión de varias cadenas independientes.

En este caso, además del operador "+", necesitaremos utilizar también la instrucción String(). Un ejemplo: para crear un objeto "unacadena" con el contenido "una frase, otra frase, otra más" deberíamos escribir: `String unacadena=String("una frase," + " otra frase," + " otra más");`.

Sea como sea, una vez creada la cadena como objeto de tipo String, ya podremos utilizar en ella todas las instrucciones listadas a continuación. En los siguientes párrafos vamos a suponer que tenemos una cadena de tipo String con el nombre de "unacadena":

unacadena.length(): devuelve la longitud de "unacadena" (es decir, el número de caracteres). No se cuenta el carácter nulo que marca los finales de cadena. No tiene parámetros.

unacadena.compareTo(): compara "unacadena" con otra que se pase como parámetro (bien de forma literal, bien mediante un objeto String). Comparar significa detectar qué cadena se ordena antes que la otra. Las cadenas se comparan desde su principio carácter a carácter utilizando el orden de sus códigos ASCII. Esto quiere decir, por ejemplo, que 'a' va antes que 'b' pero después que 'A', y las cifras van antes de las letras. Su valor de retorno será un número negativo si "unacadena" va antes que la cadena pasada como parámetro, 0 si las dos cadenas son iguales o un número positivo si "unacadena" va después que la cadena pasada como parámetro.

Si se quisiera realizar una comparación similar pero entre arrays de caracteres, se puede utilizar la instrucción **strcmp()**, la cual tiene dos parámetros (que corresponden con los dos arrays a comparar), y cuyo valor de retorno es idéntico a *unacadena.compareTo()*.

unacadena.equals(): compara si "unacadena" es igual a otra cadena, pasada como parámetro. La comparación es "case-sensitive": esto quiere decir que la cadena "hola" no es igual a "HOLA". Esta instrucción es equivalente al operador "==" para objetos String (este operador lo veremos dentro de poco). Su valor de retorno es "true" si "unacadena" es igual a la cadena especificada como parámetro, o "false" en caso contrario.

unacadena.equalsIgnoreCase(): compara si "unacadena" es igual a otra cadena, pasada como parámetro. La comparación es "case-insensitive": esto quiere decir que la cadena "hola" es igual a "HOLA". Su valor de retorno es "true" si "unacadena" es igual a la cadena especificada como parámetro, o "false" en caso contrario.

unacadena.indexOf(): devuelve la posición (un número entero) dentro de "unacadena" donde se encuentra el carácter o el principio de la cadena especificada como parámetro. Si no se encuentra nada, devuelve -1. Observar que las posiciones se numeran empezando por 0. Por defecto, la búsqueda comienza desde el principio de "unacadena", pero mediante un segundo parámetro opcional se puede indicar la posición a partir de la cual se quiere empezar a buscar. De esta manera, se puede utilizar esta instrucción para encontrar paso a paso todas las ocurrencias que existan de la cadena buscada.

unacadena.lastIndexOf(): devuelve la posición (un número entero) dentro de "unacadena" donde se encuentra el carácter o el principio de la cadena especificada como parámetro. Si no se encuentra nada, devuelve -1. Observar que las posiciones se numeran empezando por 0. Por defecto, la búsqueda comienza desde el final de "unacadena" hacia atrás, pero mediante un segundo parámetro opcional se puede indicar la posición a partir de la cual se quiere empezar a buscar (hacia atrás siempre). De esta manera, se puede utilizar esta instrucción para encontrar paso a paso todas las ocurrencias que existan de la cadena buscada.

unacadena.charAt(): devuelve el carácter cuya posición (dato entero) se haya especificado como parámetro. Las posiciones se numeran empezando por 0.

unacadena.substring(): devuelve la subcadena dentro de "unacadena" existente entre la posición inicial (especificada como primer parámetro) y la posición final (especificada como segundo parámetro opcional). La posición inicial indicada es inclusiva (es decir, el carácter que ocupa esa posición es incluido en la subcadena), pero la posición final –opcional– es exclusiva (es decir, el carácter que ocupa esa posición es el primero en no incluirse en la subcadena). Si dicha posición final no se especifica, la subcadena continúa hasta el final de "unacadena". Observar que las posiciones se numeran empezando por 0.

unacadena.replace(): sustituye una subcadena existente dentro de "unacadena" (especificada como primer parámetro) por otra (especificada como segundo), todas las veces que aparezca. También sirve para sustituir caracteres individuales. La sustitución se realiza sobre "unacadena", sobrescribiendo su valor original.

unacadena.toLowerCase(): convierte todos los caracteres de “unacadena” en minúsculas. La conversión se realiza sobre “unacadena”, sobrescribiendo su valor original.

unacadena.toUpperCase(): convierte todos los caracteres de “unacadena” en mayúsculas. La conversión se realiza sobre “unacadena”, sobrescribiendo su valor original.

unacadena.trim(): elimina todos los espacios en blanco y tabulaciones existentes al principio y al final de “unacadena”. La conversión se realiza sobre “unacadena”, sobrescribiendo su valor original.

unacadena.concat() : añade (“concatena”) al final de la cadena “unacadena” otra cadena, pasada como parámetro. Como resultado obtendremos un nuevo valor en “unacadena”: su valor original seguido de ese valor pasado por parámetros, unidos. Es equivalente al operador “+” para objetos String.

unacadena.endsWith(): chequea si “unacadena” acaba con los caracteres de otra cadena, pasada por parámetro. Su valor de retorno es “true” si “unacadena” acaba con la cadena especificada como parámetro, o “false” en caso contrario.

unacadena.startsWith() : Chequea si “unacadena” empieza con los caracteres de otra cadena, pasada por parámetro. Su valor de retorno es “true” si “unacadena” empieza con la cadena especificada como parámetro, o “false” en caso contrario.

unacadena.toCharArray(): copia una cantidad determinada de caracteres pertenecientes a “unacadena” a un array de tipo “char”. Ese array ha de ser especificado como primer parámetro, y la cantidad de caracteres a copiar allí ha de ser especificada como segundo parámetro. –de tipo “word”– . Siempre se empiezan a obtener los caracteres desde el principio de “unacadena”. No tiene valor de retorno.

unacadena.getBytes(): copia una cantidad determinada de caracteres pertenecientes a “unacadena” a un array de tipo “byte”. Ese array ha de ser especificado como primer parámetro, y la cantidad de caracteres a copiar allí ha de ser especificada como segundo parámetro. –de tipo “word”– . Siempre se empiezan a obtener los caracteres desde el principio de “unacadena”. No tiene valor de retorno.

unacadena.toInt(): si “unacadena” tiene un valor que empieza por cifras numéricas, esta instrucción es capaz de distinguirlas (descartando los posibles caracteres no numéricos posteriores) y devolver ese valor numérico transformado en un dato de tipo entero. Es decir, transforma una cadena en un número entero, si es posible. Esta instrucción no tiene parámetros.

Ejemplo 4.15: A continuación, se muestra un sketch de ejemplo donde se puede observar el comportamiento de la mayoría de instrucciones descritas anteriormente:

```
String unacadena="En un lugar de La Mancha de cuyo nombre";
String otracadena="Erase una vez";
byte pos=0;
String cadenaEntero="13asdf";
void setup(){
       Serial.begin(9600);
//Devolverá 39
       Serial.println(unacadena.length());
//Devolverá 1 (true)
       Serial.println(unacadena.endsWith("nombre"));
//Devolverá 1 (true)
       Serial.println(unacadena.startsWith("En"));
//Devolverá un número negativo ('n' va antes de 'r')
       Serial.println(unacadena.compareTo(otracadena));
//Devolverá 0 (son diferentes)
      Serial.println(unacadena.equals(otracadena));
//Devolverá 0 (son diferentes)
      Serial.println(unacadena.equalsIgnoreCase(otracadena));
//Primero devolverá 12 (primer "de") y luego 25 (segundo "de")
      pos=unacadena.indexOf("de");
      Serial.println(pos);
      Serial.println(unacadena.indexOf("de",pos+1));
//Devolverá -1 porque se empieza a buscar antes
       Serial.println(unacadena.lastIndexOf("Mancha",3));
//Devolverá 'n'
      Serial.println(unacadena.charAt(1));
//Devolverá “un”
       Serial.println(unacadena.substring(3,6));
//Devolverá “cuyo nombre”
      Serial.println(unacadena.substring(unacadena.indexOf("cuyo")));
//La palabra “apellido” sustituye a “nombre”
       unacadena.replace("nombre","apellido");
       Serial.println(unacadena);
//La letra 'I' sustituye a la primera ('E')
       unacadena.setCharAt(0,'I');
```



```
       Serial.println(unacadena);
//Toda la cadena se transforma en mayúsculas
       unacadena.toUpperCase();
       Serial.println(unacadena);
//Se eliminan los espacios en blanco en los extremos
       unacadena.trim();
       Serial.println(unacadena);
//"otracadena" se concatena con "unacadena"
       unacadena.concat(otracadena);
       Serial.println(unacadena);
//Devolverá "13asdf7" ("+" actúa como concatenador de cadenas)
      Serial.println(cadenaEntero+7);
/*La instrucción toInt() convierte la cadena "13asdf7" en el número
  entero 13, por lo que "+" ahora actúa como operador suma */
      Serial.println(cadenaEntero.toInt() +7);
}
void loop(){}
```

CREACIÓN DE INSTRUCCIONES (FUNCIONES) PROPIAS

Imaginemos que tenemos un conjunto de instrucciones que hemos de escribir repetidas veces en diferentes partes de nuestro sketch. ¿No habría alguna manera para poder invocar a ese conjunto de instrucciones mediante un simple nombre sin tener que volver a escribir todas ellas cada vez? Sí, mediante la creación de funciones. Una función es un trozo de código al que se le identifica con un nombre. De esta forma, se puede ejecutar todo el código incluido dentro de ella simplemente escribiendo su nombre en el lugar deseado de nuestro sketch.

Al crear nuestras propias funciones, escribimos código mucho más legible y fácil de mantener. Segmentar el código en diferentes funciones permite al programador crear piezas modulares de código que realizan una tarea definida. Además, una función la podemos reutilizar en otro sketch, de manera que con el tiempo podemos tener una colección muy completa de funciones que nos permitan escribir código muy rápida y eficientemente.

Resumiendo, incluir fragmentos de código en funciones tiene diversas ventajas: las funciones ayudan al programador a ser organizado (a menudo esto ayuda a conceptualizar el programa), codifican una acción en un lugar, de manera que una función solo ha de ser pensada y escrita una vez (esto también reduce las posibilidades de errores en una modificación, si el código ha de ser cambiado) y hacen más fácil la reutilización de código en otros programas (provocando que estos sean más pequeños, modulares y legibles).

Para crear una función propia, debemos “declararlas”. Esto se hace en cualquier lugar fuera de “void setup()” y “void loop()” –por tanto, bien antes o después de ambas secciones– , siguiendo la sintaxis marcada por la plantilla siguiente:

```
tipoRetorno nombreFuncion (tipo param1, tipo param2,...) {
                    // Código interno de la función
}
```

donde:

“tipoRetorno” es uno de los tipos ya conocidos (“byte”, “int”, “float”, etc.) e indica el tipo de valor que la función devolverá al sketch principal una vez ejecutada. Este valor devuelto se podrá guardar en una variable para ser usada en el sketch principal, o simplemente puede ser ignorado. Si no se desea devolver ningún dato (es decir: que la función realice su tarea y punto), se puede utilizar como tipo de retorno uno especial llamado “void” o bien no especificar ninguno. Para devolver el dato, se ha de utilizar la instrucción *return valor;*, la cual tiene como efecto “colateral” el fin de la ejecución de la función. Esto hace que normalmente la instrucción “return” sea la última en escribirse dentro del código de la función. Si la función no retorna nada (es decir, si el tipo de retorno es “void”), no es necesario escribirla.

”tipo param1, tipo param2,...” son las declaraciones de los parámetros de la función, que no son más que variables internas cuya existencia solo perdura mientras el código de esta se esté ejecutando. El número de parámetros puede ser cualquiera: ninguno, uno, dos, etc. El valor inicial de estos parámetros se asigna explícitamente en la “llamada” a la función (esto es, cuando esta es invocada dentro del sketch principal), pero este valor puede variar dentro de su código interno. En todo caso, al finalizar la ejecución de la función, todos sus parámetros son destruidos de la memoria del microcontrolador.

Fijarse que el código interno de la función está delimitado por las llaves de apertura y cierre.

Ejemplo 4.16: Veamos un ejemplo de creación de una función que devuelve el resultado de multiplicar dos números pasados como parámetros:


```
int multiplicar(int x, int y){
/*La variable "resultado", al estar declarada dentro de la función
"multiplicar", no existe fuera de ella. */
	int resultado;
/*"x" e "y" valdrán lo que se ponga en cada invocación que se realice
de la función "multiplicar()" en el sketch principal (en este caso,
"x"="primernumero" e "y"="segundonumero")*/
	resultado = x * y ;
/*El tipo de la variable "resultado" ha de coincidir con el tipo de
retorno ("int") */
	return resultado;
}
void setup(){
	Serial.begin(9600);
}
void loop(){
	int primernumero=2;
	int segundonumero=3;
	int recojoresultado;
/*Los valores pasados como parámetros han de ser del mismo tipo que
los especificados en la declaración de la función. Por otro lado, a
la variable "recojoresultado", propia del sketch principal, se le
asignará el valor que retorna la función "multiplicar()" gracias a su
variable interna "resultado"; lógicamente,"recojoresultado" y
"resultado" han de ser del mismo tipo también */
	recojoresultado=multiplicar(primernumero,segundonumero);
	Serial.println(recojoresultado);
}
```

Uno podría pensar que el uso de parámetros en una función no es necesario porque mediante variables globales cualquier función podría acceder a los valores necesarios para trabajar. Es decir, podríamos haber hecho algo así:

```
int primernumero=2;
int segundonumero=3;
int multiplicar(){
	int resultado;
	resultado = primernumero * segundonumero ;
	return resultado;
}
void setup(){
	Serial.begin(9600);
}
void loop(){
```

```
        int recojoresultado;
        recojoresultado=multiplicar();
        Serial.println(recojoresultado);
}
```

En el código anterior, tanto "primernumero" como "segundonumero" se han declarado como variables globales y la función *multiplicar()* no ha necesitado por tanto ningún parámetro. No obstante, el problema de usar variables globales en vez de pasar parámetros a la función es la pérdida de elegancia y flexibilidad. ¿Qué pasaría si además de querer multiplicar 2x3 quisiéramos multiplicar 4x7? En el caso de utilizar parámetros, sería tan sencillo como pasar los nuevos valores directamente como parámetros (así: *multiplicar(4,7);*) para tener lo que queremos. En cambio, al utilizar variables globales necesitaríamos cambiar sus valores cada vez que quisiéramos multiplicar diferentes números (o tener varias variables globales para cada número que deseáramos multiplicar, algo bastante inviable), con lo que podríamos alterar el comportamiento de otras partes de nuestro código y ser una fuente de errores.

Finalmente, indicar que aunque hasta ahora hemos ido llamando "instrucción" a los diferentes comandos del lenguaje Arduino (algunos dentro de objetos y otros no), en realidad, todas estas instrucciones son también funciones. Funciones que permiten invocar un conjunto de código escrito en lenguaje C invisible para nosotros. Incluso lo que hemos venido llamando la "sección" "void setup()" y la "sección" "void loop()" también son funciones (sin parámetros y sin valor de retorno, como se puede ver).

BLOQUES CONDICIONALES

Los bloques "if" y "if/else"

Un bloque "if" sirve para comprobar si una condición determinada es cierta ("true",1) o falsa ("false",0). Si la condición es cierta, se ejecutarán las instrucciones escritas en su interior (es decir, dentro de las llaves de apertura y cierre). Si no se cumple, puede no pasar nada, o bien, si existe tras el bloque "if" un bloque "else" (opcional), se ejecutarán las instrucciones escritas en el interior de ese bloque "else". Es decir, si solo escribimos el bloque "if", el sketch tendrá respuesta solamente para cuando sí se cumple la condición; pero si además escribimos un bloque "else", el sketch tendrá respuesta para cuando sí se cumple la condición y para cuando no se cumple también. En general, la sintaxis del bloque "if/else" es:

```
if (condición) {
      //Instrucciones –una o más– que se ejecutan si la condición es cierta –"true",1–
} else {
      //Instrucciones –una o más– que se ejecutan si la condición es falsa –"false",0–
}
```

Tanto si se ejecutan las sentencias del bloque "if" como si se ejecutan las sentencias del bloque "else", cuando se llega a la última línea de esa sección (una u otra), se salta a la línea inmediatamente posterior a su llave de cierre para continuar desde allí la ejecución del programa.

Hemos dicho que el bloque "else" es opcional. Si no lo escribimos, el "if" quedaría así:

```
if (condición) {
      //Instrucciones –una o más– que se ejecutan si la condición es cierta –"true",1–
}
```

En este caso, si la condición fuera falsa, el interior del "if" no se ejecutaría y directamente se pasaría a ejecutar la línea inmediatamente posterior a su llave de cierre (por lo que, tal como ya hemos comentado, el programa no hace nada en particular cuando la condición es falsa).

También existe la posibilidad de incluir una o varias secciones "else if", siendo en este caso también opcional el bloque "else" final. Esta construcción tendría la siguiente sintaxis (puede haber todos los "else if" que se deseen):

```
if (condición) {
     //Instrucciones –una o más– que se ejecutan si la condición es cierta
} else if (otra_condición) {
               /*Instrucciones –una o más– que se ejecutan si la condición
                   del anterior "if" es falsa pero la actual es cierta */
} else if (otra_condición) {
               /*Instrucciones –una o más– que se ejecutan si la condición
                   del anterior "if" es falsa pero la actual es cierta */
}  else {
    //Instrucción(es) que se ejecutan si todas las condiciones anteriores eran falsas
}
```

Es posible anidar bloques "if" uno dentro de otro sin ningún límite (es decir, se pueden poner más bloques "if" dentro de otro bloque "if" o "else", si así lo necesitamos).

Ahora que ya sabemos las diferentes sintaxis del bloque "if", veamos qué tipo de condiciones podemos definir entre los paréntesis del "if". Lo primero que debemos saber es que para escribir correctamente en nuestro sketch estas condiciones necesitaremos utilizar alguno de los llamados operadores de comparación, que son los siguientes.

Operadores de comparación	
==	Comparación de igualdad
!=	Comparación de diferencia
>	Comparación de mayor que
>=	Comparación de mayor o igual que
<	Comparación de menor que
<=	Comparación de menor o igual que

Ejemplo 4.17: Sabiendo esto, ya podemos escribir todas las condiciones que deseemos. A continuación, se muestra un código de ejemplo.

```
int numero;
void setup(){
	Serial.begin(9600);
}
void loop(){
	if (Serial.available() > 0){
	//El n° introducido ha de estar en el rango del tipo "int"
		numero=Serial.parseInt();
		if (numero == 23){
			Serial.println("Numero es igual a 23");
		} else if (numero < 23) {
			Serial.println("Numero es menor que 23");
		} else {
			Serial.println("Numero es mayor que 23");
		}
	}
}
```


En el código anterior aparece un primer “if” que comprueba si hay datos en el buffer de entrada del chip TTL-UART pendientes de leer. Si es así (es decir, si el valor devuelto por *Serial.available()* es mayor que 0), se ejecutarán todas las instrucciones en su interior. En cambio, si la condición resulta ser falsa (es decir, si *Serial.available()* devuelve 0 y por tanto no hay datos que leer), fijarse que la función “loop()” no ejecuta nada.

En el momento que la condición sea verdadera, lo que tenemos dentro del primer “if” es la función *Serial.parseInt()* que reconoce y extrae de todo lo que se haya enviado a través del canal serie (por ejemplo, usando el “Serial monitor”) un número entero, asignándolo a la variable “numero”. Y aquí es cuando llega otro “if” que comprueba si el valor de “numero” es igual a 23. Si no es así, se comprueba entonces si su valor es menor de 23. Y si todavía no es así, solo queda una opción: que sea mayor de 23. Al ejecutar este sketch a través del “Serial monitor”, lo que veremos es que cada vez que enviemos algún dato a la placa, esta nos responderá con alguna de las tres posibilidades.

En general, el “if” es capaz de comparar números decimales también: en el ejemplo anterior, si “número” es una variable “float” y sustituimos *Serial.parseInt()* por *Serial.parseFloat()* obtendríamos el mismo comportamiento. En el caso de las cadenas, no obstante, hay que tener en cuenta que solamente se pueden utilizar los operadores de comparación cuando ambas cadenas han sido declaradas como objetos String: si son arrays de caracteres no se pueden comparar. En este sentido, comparar si una cadena es mayor o menor que otra simplemente significa evaluar las cadenas en orden alfabético carácter tras carácter (por ejemplo, “alma” sería menor que “barco”, pero “999” es mayor que “1000” ya que el carácter “9” va después del “1”). Recalcar que las comparaciones de cadenas son case-sensitive (es decir, el dato de tipo String “hola” no es igual que el dato de tipo String “HOLA”).

Ejemplo 4.18: Como demostración de lo explicado en el párrafo anterior, en el siguiente sketch se ilustra el uso del operador de igualdad, el cual en este caso resulta funcionalmente equivalente a escribir la expresión `cad1.equals(cad2)`.

```
String cad1="hola";
String cad2="hola";
void setup(){
       Serial.begin(9600);
}
void loop(){
       if (cad1 == cad2){
```

```
                Serial.println("La cadena es igual");
        } else {
                Serial.println("La cadena es diferente");
        }
}
```

Por otro lado, es bastante probable encontrar en bastantes códigos expresiones tales como *if(mivariable)* en vez de *if(mivariable!=0)*, por ejemplo. Es decir, "ifs" que solo incluyen una variable, pero no la condición para evaluarla. Esta forma de escribir los "ifs" es simplemente un atajo: si en el "if" solo aparece una variable o expresión sin ningún tipo de comparación, es equivalente a comprobar si dicha variable o expresión es "true" (es decir, si es diferente de 0). Si se diera el caso de que en el "if" aparece tan solo una función, sería equivalente a comprobar si el valor que devuelve esa función es, también, diferente de 0.

Seguramente sorprenderá que el operador de igualdad ("==") sea un doble igual. Esto es debido a que el signo igual individual ("=") representa el operador de asignación. Por lo tanto, ambos símbolos tienen un significado totalmente diferente, y pueden dar muchos problemas para quien se despiste. Por ejemplo, si se escribe por error una condición tal como *if(x = 10)* , lo que se está haciendo es asignar el valor 10 a la variable "x", y esta orden siempre hace que la "condición" sea verdadera, porque lo que hace Arduino (tal como acabamos de explicar en el párrafo anterior) es comprobar si el valor asignado a "x" (10) es "true" (es decir, diferente de 0), cosa que es evidentemente cierto siempre. Lógicamente, lo correcto hubiera sido escribir *if(x==10).*

Además de los operadores de comparación, en las comparaciones también se pueden utilizar los operadores booleanos (también llamados lógicos), usados para encadenar dos o más comprobaciones dentro de una condición. Son los siguientes:

Operadores lógicos

&&	Comprueba que las dos condiciones sean ciertas	(Operador AND)
||	Comprueba que, al menos, una de dos condiciones sea cierta	(Operador OR)
!	Comprueba que no se cumpla la condición a la que precede	(Operador NOT)

El operador "AND" obliga a que todas las condiciones de dentro del paréntesis del "if" sean ciertas: si hay una sola que ya no lo es, el conjunto total tampoco lo será. Al operador "OR", en cambio, le basta con que, de entre todas las

condiciones, una sola ya sea cierta para que el conjunto total lo sea también. El operador NOT cambia el estado de la condición que le sigue: si esa condición era cierta pasa a ser falsa, y viceversa.

Ejemplo 4.19: El siguiente sketch ilustra lo anterior:

```
void setup(){
	byte x=50;
	Serial.begin(9600);
	if (x >=10 && x <=20){
		Serial.println("Frase 1");
	}
	if (x >=10 || x <=20) {
		Serial.println("Frase 2");
	}
	if (x >=10 && !(x<=20)) {
		Serial.println("Frase 3");
	}
}
void loop(){}
```

Si ejecutamos el código anterior, veremos que solamente se muestran la “Frase 2” y la “Frase 3”. Esto es debido a que la condición del primer “if” es falsa: en él se comprueba si la variable “a” es mayor que 10 Y A LA VEZ si es menor de 20. En cambio, la condición del segundo “if” es cierta: en él se comprueba si la variable “a” es mayor que 10 O BIEN menor de 20: como ya se cumple una de las condiciones –la primera–, la condición total ya es cierta valga lo que valga el resto de condiciones presentes. La condición del tercer “if” también es cierta: en él se comprueba si la variable “a” es mayor que 10 y a la vez, gracias al operador “!”, que NO sea menor de 20.

Otra funcionalidad muy útil de los “ifs” es el poder utilizar los paréntesis dentro de una condición para establecer el orden de comparación de varias expresiones. Siempre se comparan primero las expresiones situadas dentro de los paréntesis, o en el más interior en el caso de que haya varios grupos de paréntesis anidados. Por ejemplo, podríamos tener una condición como esta: *if (x!=0 || (y>=100 && y <=200)).* En este ejemplo, primero se comprueba lo que hay dentro del paréntesis, es decir, que “y” sea mayor o igual que 100 y que además sea menor o igual que 200. Eso, o que “x” sea diferente de 0. El anidamiento de condiciones puede ser todo lo complejo que uno quiera, pero conviene asegurarse de que realmente se está comprobando lo que uno quiere, cosa que a veces, con expresiones muy largas, es difícil.

El bloque "switch"

Como se ha podido ver en el apartado anterior, los bloques "else if" se tienen en cuenta siempre y cuando las condiciones evaluadas hasta entonces hayan sido falsas, y la condición del propio "else if" sea la cierta. Es decir, un bloque *if(condicion1){}else if(condicion2){}* se puede leer como "si ocurre condicion1, haz el interior del primer if, y si no, mira a ver si ocurre condicion2, y (solo) si es así, haz entonces el interior del elseif". Esta es una manera válida de hacer comprobaciones de condiciones múltiples, pero existe otra forma más elegante, cómoda y fácil de hacer lo mismo: utilizar el bloque "switch". Su sintaxis es la siguiente:

```
switch (expresión) {
        case valor1:
         //Instrucciones que se ejecutarán cuando "expresión" sea igual a "valor1"
        break;
        case valor2:
        //Instrucciones que se ejecutarán cuando "expresión" sea igual a "valor2"
        break;
        /*Puede haber los "case" que se deseen,
       y al final una sección "default" (opcional)*/
        default:
       //Instrucciones que se ejecutan si no se ha ejecutado ningún "case" anterior
}
```

Un bloque "switch" es como una especie de "if else" escrito más compactamente. Como se puede ver, consta en su interior de una serie de secciones "case" y, opcionalmente, de una sección "default". Nada más llegar a la primera línea del "switch", primero se comprueba el valor de la variable o expresión que haya entre sus paréntesis (siguiendo las mismas reglas y operadores posibles usados en un "if" estándar). Si el resultado es igual al valor especificado en la primera sección "case", se ejecutarán las instrucciones del interior de la misma y se dará por finalizado el "switch", continuando la ejecución del sketch por la primera línea después de la llave de cierre. En caso de no ser igual el resultado de la expresión a lo especificado en el primer "case" se pasará a comprobarlo con el segundo "case" , y si no con el tercero, etc. Por último, si existe una sección "default" (opcional) y el resultado de la expresión no ha coincidido con ninguna de las secciones "case", entonces se ejecutarán las sentencias de la sección "default".

En una sección "case" el valor a comparar tal solo puede ser de tipo entero. Otros lenguajes de programación permiten comparar otros tipos de datos, o

comparar rangos de valores en un solo "case", o comparar más de un valor individual en un solo "case", etc., pero el lenguaje Arduino no.

Hay que hacer notar que una vez ejecutada una de las secciones "case" de un bloque "switch" ya no se ejecutan más secciones "case", aunque estas también dispongan del resultado correcto de la expresión evaluada: esto es así gracias a la instrucción "break;", que comentaremos en breve.

No es necesario ordenar las secciones "case" según sus valores (de menor a mayor, o de mayor a menor), pero sí es imprescindible que la sección "default" (en caso de haberla) sea la última sección, y no puede haber más que una.

Es posible anidar sentencias "switch" sin ningún límite (es decir, se pueden poner nuevas sentencias "switch" dentro de una sección "case").

Ejemplo 4.20: Un código bastante sencillo que muestra su uso es el siguiente:

```
void setup(){
        byte x=50;
        Serial.begin(9600);
        switch (x) {
                case 20:
                        Serial.println("Vale 20 exactamente");
                break;
                case 50:
                        Serial.println("Vale 50 exactamente");
                break;
                default:
        Serial.println("No vale ninguna de los valores anteriores");
        }
}
void loop(){}
```

BLOQUES REPETITIVOS (BUCLES)

El bloque "while"

El bloque "while" ("mientras", en inglés) es un bloque que implementa un bucle; es decir, repite la ejecución de las instrucciones que están dentro de sus llaves de apertura y cierre. ¿Cuántas veces? No hay un número fijo: se repetirán mientras la condición especificada entre sus paréntesis sea cierta ("true",1). Su sintaxis es muy sencilla:

```
while (condición) {
        //Instrucciones que se repetirán mientras la condición sea cierta –"true",1–
}
```

La condición escrita entre paréntesis sigue las mismas reglas y puede utilizar los mismos operadores que hemos visto con el bloque “if”. Nada más llegar a la línea donde aparece escrita esta condición, esta se comprobará; si resulta cierta, se ejecutarán las sentencias interiores, y si no, la ejecución del programa continuará a partir de la línea siguiente a la llave de cierre. En el primer caso (cuando la condición es cierta), una vez ejecutadas todas las instrucciones del interior del bloque “while”, se volverá a comprobar de nuevo la condición, y si esta continúa siendo cierta, se realizará otra iteración (es decir: se volverán a ejecutar las sentencias interiores). Cuando se llegue de nuevo al final de esas instrucciones anteriores, se volverá a evaluar la condición, y si sigue siendo cierta, se volverán a ejecutar. Este proceso continuará hasta que, en un momento dado, al comprobarse la condición del “while”, esta resulte falsa.

Si se llega por primera vez a una sentencia “while” y la condición resulta falsa directamente, no se ejecutarán las sentencias interiores ninguna vez. Este detalle es importante tenerlo en cuenta.

Las sentencias interiores a un bucle “while” pueden ser tantas como se quieran y de cualquier tipo, incluyendo, por supuesto, nuevos bucles “while”.

Ejemplo 4.21: En el siguiente ejemplo de muestra, se puede observar cómo en el “Serial monitor” los primeros 50 mensajes que aparecerán indicarán que la variable es menor de 50 porque se está ejecutando el interior del “while”. Pero en el momento que la variable sea mayor (porque en cada iteración del bucle se le va aumentando en una unidad su valor), la condición del “while” dejará de ser cierta y saltará de allí, mostrando entonces el mensaje (ya de forma infinita) de que la variable es mayor que 50:

```
byte x=1;
void setup(){
        Serial.begin(9600);
}
void loop(){
        while (x <= 50){
                Serial.println("Es menor de 50");
                x=x+1;
        }
```



```
	Serial.println("Es mayor que 50");
}
```

Ejemplo 4.22: Una aplicación práctica del bucle "while" es el poder recibir a través del canal serie (u otro sistema de comunicación diferente) varios caracteres seguidos uno tras otro, e irlos concatenando (por ejemplo, mediante la función *unacadena.concat()*), para lograr así generar y reconocer palabras y frases enteras. Esto nos puede venir muy bien para comunicarnos con nuestra placa Arduino mediante comandos completos en vez de tan solo con simples caracteres, de tal forma que podamos construir un lenguaje de control propio (incluso con parámetros).

```
char caracter;
String comando;
void setup(){
	Serial.begin(9600);
}
void loop(){
/*Voy leyendo carácter a carácter lo que se recibe por el canal serie
(mientras llegue algún dato allí), y los voy concatenando uno tras
otro en una cadena. En la práctica, si usamos el "Serial monitor" el
bucle while acabará cuando pulsemos Enter. El delay es conveniente
para no saturar el canal serie y que la concatenación se haga de
forma ordenada. */
  while (Serial.available()>0){
    caracter= Serial.read();
    comando.concat(caracter);
    delay(10);
  }
/*Una vez ya tengo la cadena "acabada", compruebo su valor y hago que
la placa Arduino reaccione según sea este. Aquí podríamos hacer lo
que quisiéramos: si el comando es "tal", enciende un LED, si es cual,
mueve un motor… y así*/
  if (comando.equals("hola") == true){
        Serial.println("El comando es hola");
  }
  if (comando.equals("adiós")== true){
        Serial.println("El comando es adiós");
  }
//Limpiamos la cadena para volver a recibir el siguiente comando
  comando="";
}
```

Otra aplicación práctica del bucle "while" es la eliminación de los datos en el buffer de entrada del chip TTL-UART que no queramos, de manera que se quede este limpio de "basura". Esto lo podríamos hacer simplemente así:

```
while (Serial.available >0){
      Serial.read();
}
```

El bloque "do"

El bloque "do" tiene la siguiente sintaxis:

```
do {
      //Instrucciones que se repetirán mientras la condición sea cierta –"true",1–
} while (condición)
```

El bucle "do" funciona exactamente igual que el bucle "while", con la excepción de que la condición es evaluada después de ejecutar las instrucciones escritas dentro de las llaves. Esto hace que las instrucciones siempre sean ejecutadas como mínimo una vez aun cuando la condición sea falsa, porque antes de llegar a comprobar esta, las instrucciones ya han sido leídas (a diferencia del bucle "while", donde si la condición ya de entrada era falsa las instrucciones no se ejecutaban nunca).

El bloque "for"

La diferencia entre un bucle "while" (o "do") y un bucle "for" está en que en el primero el número de iteraciones realizadas depende del estado de la condición definida, pero en un bucle "for" el número de iteraciones se puede fijar a un valor exacto. Por tanto, usaremos el bucle "for" para ejecutar un conjunto de instrucciones (escritas dentro de llaves de apertura y cierre) un número concreto de veces. La sintaxis general del bucle "for" es la siguiente:

```
for (valor_inicial_contador;condicion_final;incremento){
      //Instrucciones que se repetirán un número determinado de veces
}
```

Tal como se observa, entre paréntesis se deben escribir tres partes diferentes, separadas por puntos y coma. Estas tres partes son opcionales (pueden omitirse cualquiera de ellas) y son las siguientes:

Valor inicial del contador: en esta parte se asigna el valor inicial de una variable entera que se utilizará como contador en las iteraciones del bucle. Por ejemplo, si allí escribimos *x=0* , se fijará la variable "x" a cero al inicio del bucle. A partir de entonces, a cada repetición del bucle, esta variable "x" irá aumentando (o disminuyendo) progresivamente de valor.

Condición final del bucle: en esta parte se especifica una condición (del estilo de las utilizadas en un bucle "while"). Justo antes de cada iteración se comprueba que sea cierta para pasar a ejecutar el grupo de sentencias internas. Si la condición se evalúa como falsa, se finaliza el bucle "for", continuando el programa tras su llave de cierre. Por ejemplo, si allí escribimos *x<10* , el grupo interior de sentencias se ejecutará únicamente cuando la variable "x" valga menos de 10 (es decir, mientras x<10 sea cierto).

Incremento del contador: en la última de las tres partes es donde se indica el cambio de valor que sufrirá al inicio de cada iteración del bucle la variable usada como contador. Este cambio se expresa con una asignación. Por ejemplo, la sentencia *x=x+1* le sumará 1 a la variable "x" antes de cada nueva iteración del bucle, por lo que en realidad estaríamos haciendo un contador que aumenta de uno en uno a cada repetición. Este cambio se efectúa justo antes de comprobar la condición de final del bucle.

Ejemplo 4.23: Veamos mejor un código de ejemplo:

```
byte x;
void setup(){
	Serial.begin(9600);
	for (x=0;x<10;x=x+1){
		Serial.println(x);
	}
}
void loop(){}
```

Si observamos el resultado mostrado por el "Serial monitor", veremos que aparece una lista de números del 0 al 9. ¿Por qué? Porque cuando en el sketch se llega a la sentencia "for" primero se inicializa el contador (x=0) y seguidamente se comprueba la condición. Como efectivamente, x<10, se ejecutará la –única en este caso– sentencia interior, que muestra el propio valor del contador por el "Serial monitor". Justo después, se realiza el incremento (x=x+1), volviéndose seguidamente a comprobar la condición. Si "x" (que ahora vale 1) sigue cumpliendo la condición (sí,

porque 1<10), se volverá a ejecutar la instrucción interna, mostrándose ahora el valor "1" en el "Serial monitor". Justo después de esto se volverá a realizar el incremento (valiendo x ahora por tanto 2) y se volverá a comprobar la condición, y así y así hasta que llegue un momento en el que la condición resulte falsa (cuando "x" haya incrementado tanto su valor que ya sea igual o mayor que 10), momento en el cual se finalizará el "for" inmediatamente.

Es importante recalcar que el número 10 no se imprimirá, porque, tal como hemos comentado, primero se incrementa el contador y luego se hace la comprobación. Nada más acabar de ejecutar la última iteración tendremos que "x" vale 9; entonces se incrementa a 10 y seguidamente se comprueba si 10 es menor que 10. Al ser esto falso, se sale del "for", por lo que cuando "x" vale 10 ya no tiene la oportunidad de ser imprimida. Si hubiéramos querido imprimir el 10, tendríamos que haber escrito como condición *x <=10*, en cuyo caso el "for" se hubiera repetido 11 veces.

Evidentemente, no es obligatorio que la inicialización del contador empiece por 0, ni que el incremento se realice de uno en uno (ni tan siquiera que sea un incremento: puede ser un decremento, como *x=x-1* o similar).

Como se ha mencionado, las tres partes dentro del paréntesis de la definición del bucle son opcionales. De hecho, si se omiten las tres (es decir, si se escribiera *for (;;)*) estaríamos escribiendo un bucle infinito, tal como la función "loop()".

Las sentencias interiores de un bucle "for" pueden ser tantas como se quieran y de cualquier tipo, incluyendo, por supuesto, nuevos bucles "for" (lo que se llama "for anidados").

Si se declara una variable dentro de un bucle "for", solamente existirá mientras este bucle "for" se esté ejecutando.

Ejemplo 4.24: Veamos otro ejemplo de uso del bucle "for", que nos será muy útil es próximos proyectos:

```
byte suma=0;
byte x;
void setup(){
       Serial.begin(9600);
       for (x=0;x<7;x=x+1){ //Sumo 7 veces el número 5
             suma= suma + 5;
       }
```



```
      Serial.println(suma);
      Serial.println(suma/7); //Esto es la media
}
void loop(){}
```

En el código anterior se puede ver una manera bastante común de realizar sumas de muchos valores (y su media). Muchas veces tendremos la necesidad de calcular estas operaciones, y la mecánica siempre es la misma: en este ejemplo hemos utilizado una variable (en este caso, llamada “suma”), que empieza valiendo 0. En la primera repetición del bucle “for” se le asigna el valor del primer dato a sumar; en la segunda repetición a ese valor de “suma” se le suma el segundo valor (por lo que en ese momento “suma” vale la suma de los dos primeros valores). En la tercera repetición se le añade el tercer valor a esa suma parcial (por lo que en ese momento “suma” vale la suma de los tres primeros valores), en la cuarta repetición el cuarto valor, y así y así hasta que se acaba de recorrer el bucle “for” y todos los datos se han ido añadiendo uno tras otro hasta conseguir la suma total. Finalmente, muestro el resultado, y muestro también la media, que no es más que esa suma total entre el número de datos introducidos en la suma.

Ejemplo 4.25: Otra aplicación práctica y concreta del bucle “for” es utilizarlo para recorrer los elementos de un array, uno por uno. El contador del bucle es usado como el índice (la posición) de cada elemento del array: inicialmente este contador se sitúa en el primer elemento y va aumentando en cada iteración hasta llegar al último. Por ejemplo, suponiendo que hemos declarado previamente un array de 5 elementos de tipo “char” llamado “miarray”, podríamos hacer lo siguiente para asignar un valor (el mismo, en este caso) a cada uno de ellos y seguidamente mostrarlo en el “Serial monitor” junto con su índice correspondiente:

```
for (i=0; i <5;i=i+1){
      miarray[i]='a';
      Serial.print(i);
      Serial.println(miarray[i]);
}
```

Observar en el ejemplo anterior que el contador empieza desde 0 (igual que la numeración de las posiciones de los elementos en un array) y que el bucle realizará cinco iteraciones, valiendo “i” de 0 a 4 (que corresponden con los cinco elementos del array). En cada iteración, se asigna el valor 'a' al elemento correspondiente (primero al 0, luego al 1, etc), seguidamente se imprime el valor de “i” (0,1, etc.) y finalmente se imprime el valor recién asignado a ese elemento (que es para todos 'a').

Por otro lado, al ser la asignación de tipo *x=x+1* muy habitual en la tercera parte de la definición del bucle "for" (y en general, en todos nuestros sketches), los programadores de Arduino suelen escribir esta asignación concreta de una manera alternativa, más compacta y rápida, mediante el uso de unos signos especiales, que pertenecen a un conjunto de símbolos llamados "operadores compuestos" cuya función es precisamente la de facilitar la escritura (y lectura) de código. Estos operadores no son imprescindibles, porque lo que hacen se puede realizar con otros operadores (como los aritméticos: +,-,*,/, etc.) pero se utilizan mucho. A continuación, se presenta una tabla de "equivalencia" –incompleta– entre el uso de algunos de los operadores compuestos existentes en el lenguaje Arduino y su significado.

Operadores compuestos

x++	Equivale a x=x+1 (Al operador "++" se le llama operador "incremento")
x--	Equivale a x=x-1 (Al operador "--" se le llama operador "decremento")
x+=3	Equivale a x=x+3
x-=3	Equivale a x=x-3
x*=3	Equivale a x=x*3
x/=3	Equivale a x=x/3

Sabido esto, ahora podremos escribir por ejemplo un bucle "for" así: *for(x=0;x<10;x++){}*, que no es más que una forma más compacta de decir lo mismo que esto: *for(x=0;x<10;x=x+1){}*

Las instrucciones "break" y "continue"

La instrucción "break" y la instrucción "continue" están muy relacionadas con los bucles (ya sean del tipo "while" o "for"). Observar que ninguna de ellas incorpora paréntesis, pero como cualquier otra instrucción, en nuestros sketches deben ser finalizadas con un punto y coma.

La instrucción "break" debe estar escrita dentro de las llaves que delimitan las sentencias internas de un bucle, y sirve para finalizar este inmediatamente. Es decir, esta instrucción forzará al programa a seguir su ejecución a continuación de la llave de cierre del bucle. En caso de haber varios bucles anidados (unos dentro de otros), la sentencia "break" saldrá únicamente del bucle más interior de ellos.

La instrucción "continue" también debe estar escrita dentro de las llaves que delimitan las sentencias internas de un bucle y sirve para finalizar la iteración actual y comenzar inmediatamente con la siguiente. Es decir, esta instrucción forzará al programa a "volver para arriba" y comenzar la evaluación de la siguiente iteración aun cuando todavía queden instrucciones pendientes de ejecutar en la iteración actual. En caso de haber varios bucles anidados (unos dentro de otros) la sentencia "continue" tendrá efecto únicamente en el bucle más interior de ellos.

Ejemplo 4.25: Vamos a ver estas sentencias con un ejemplo:

```
byte x;
void setup(){
	Serial.begin(9600);
	for(x=0;x<10;x++){
		if (x==4){
			break;
		}
		Serial.println(x);
	}
}
void loop(){}
```

En el código anterior, se puede ver cómo el valor 4 de "x" ya no se llega a imprimir porque la instrucción *break* interrumpe la ejecución del bucle entero. Como tras el "for" la función "setup()" ya no tiene más código, no se ve nada más.

¿Qué pasaría si sustituimos la instrucción *break* por la sentencia *continue*, dejando el resto del código anterior exactamente igual? Que veríamos una lista de números desde el 0 hasta el 9, excepto precisamente el 4. Esto es así porque la instrucción *continue* interrumpe la ejecución de la iteración en la cual "x" es igual a 4 (y por tanto, la instrucción *Serial.println()* correspondiente no se llega a ejecutar) pero continúa con la siguiente iteración del bucle de forma normal, en la cual a "x" se le asigna el valor 5.

5 LIBRERÍAS ARDUINO

LAS LIBRERÍAS OFICIALES

Ya sabemos que una librería sirve para proporcionar funcionalidad extra en nuestros sketches (ofreciendo la posibilidad de manipular datos en diferentes formatos, de gestionar protocolos de comunicación diversos, de interactuar con hardware variado, etc.). A continuación, se hará un breve resumen de las librerías oficiales que se instalan por defecto junto con el IDE oficial de Arduino.

Recordemos que para poder utilizar las funciones que proveen, la librería en cuestión primero se ha de importar, o bien mediante la opción "Sketch"->"Import library" del menú del IDE, o bien incluyendo al principio del código del sketch la sentencia *#include* pertinente.

Librería LiquidCrystal

Permite controlar pantallas de cristal líquido (LCDs) basadas en el chip HD44780 de Hitachi (o compatibles, como el KS0066 de Samsung o el ST7065C de Sitronix, entre otros). Este modelo de chip se encuentra en la mayoría de LCDs de caracteres. La librería puede trabajar tanto en el modo 4-bit como en 8-bit (es decir: puede utilizar tanto 4 como 8 líneas de datos). Estudiaremos su uso en un apartado posterior de este capítulo.

Librería EEPROM

Permite leer y escribir datos en la memoria EEPROM del microcontrolador. Recordemos que en esta memoria puede mantener grabados los datos aunque la

placa deje de recibir alimentación eléctrica (o se resetee). Los datos se guardan en forma de conjunto de bytes, así que en realidad pueden ser de cualquier tipo. La mayor limitación que hay que tener en cuenta en este tipo de memorias es la cantidad de veces que se pueden leer o escribir datos en ella: según Atmel, la EEPROM del ATmega328P es capaz de soportar 100000 ciclos de lectura/escritura.

¿Para qué necesitaríamos usar esta memoria? Por ejemplo, para almacenar un número de serie único o la fecha de fabricación de algún proyecto comercial basado en Arduino (que se podría mostrar en una LCD, por ejemplo). O para contar algún tipo de evento sin permitir al usuario resetear el conteo. Estudiaremos su uso en un apartado posterior de este capítulo.

Librería SD

Permite leer y escribir datos en una tarjeta SD (o microSD) acoplada a un zócalo de algún shield (como por ejemplo el Arduino Ethernet Shield) o módulo específico. Las tarjetas SD son muy útiles para almacenar ficheros tales como audio, vídeo o imágenes, o bien datos textuales obtenidos de diferentes sensores, ya que ofrecen un sistema de grabación mucho mayor que la memoria EEPROM.

La librería SD puede utilizar tarjetas de tipo SDSC o SDHC, y está basada en la librería SdfatLib (http://code.google.com/p/sdfatlib), la cual puede trabajar con tarjetas formateadas en los sistemas de ficheros FAT16 y también FAT32. La comunicación entre la tarjeta SD y el microcontrolador se establece mediante SPI a través de los pines digitales 11, 12, 13 de la placa Arduino UNO, además de otro pin usado como SS (pin de activación), el cual suele ser el 10. Estudiaremos el uso de esta librería en un apartado posterior de este capítulo.

Librería Ethernet

Permite conectar el Arduino Ethernet Shield o bien la placa Arduino Ethernet (o similares) a una red Ethernet (TCP/IP). Se puede configurar para que la placa actúe como servidor (es decir, que de forma permanente e ininterrumpida escuche y acepte peticiones de otros dispositivos de la red que soliciten algún tipo de servicio o dato ofrecido por ella) o bien como cliente (es decir, que sea la placa la que solicite puntualmente esos servicios o datos a otro dispositivo de red). Esta librería soporta hasta un total cuatro conexiones concurrentes (entrantes –es decir, en “modo servidor”–, salientes –es decir, en “modo cliente”– o una combinación de estas). Estudiaremos su uso en el capítulo 8.

Librería Firmata

Permite comunicar la placa con programas ejecutados en un computador mediante una conexión de tipo serie. Firmata (http://firmata.sourceforge.net) es un protocolo genérico y sencillo de comunicación entre microcontroladores y software informático. Si una aplicación de computador es compatible con Firmata, podrá enviar y recibir datos desde y hacia la placa Arduino de una forma muy cómoda. Firmata está integrado en múltiples lenguajes de programación para precisamente poder desarrollar software compatible con él. De todas formas, esta librería no la estudiaremos en este libro: si se desea aprender sobre ella, se puede consultar su página oficial y también la documentación de referencia de la página oficial de Arduino: http://arduino.cc/en/Reference/Firmata

Librería SPI

Permite comunicar mediante el protocolo SPI la placa Arduino (que actúa siempre como "maestro") con dispositivos externos (que actuarían como "esclavos"). Tal como ya se ha comentado, la comunicación se establece mediante los pines MOSI (línea de envío de datos del maestro al esclavo, ubicada en la placa Arduino UNO en el pin digital 11), MISO (línea de envío de datos del esclavo al maestro, ubicada en el pin digital 12), SCK (línea de reloj, ubicada en el pin digital 13) y el pin SS (línea de selección del esclavo, ubicada en el pin digital 10, normalmente, aunque puede variar, como pasa por ejemplo con la placa Arduino Ethernet).

Debido a que esta librería es capaz de controlar hasta el mínimo detalle el proceso de comunicación establecido con los periféricos SPI, es relativamente compleja de utilizar ya que se necesitan ciertos conocimientos avanzados de electrónica para poder aprovecharla en todas sus posibilidades. Es por esto que no la estudiaremos en este libro. ¿Esto quiere decir que no podremos utilizar dispositivos SPI en nuestros proyectos? ¡No! Afortunadamente, existen muchas librerías de terceros que ocultan la complejidad interna de la librería SPI y ofrecen al usuario una serie de instrucciones mucho más sencillas de aprender y utilizar. Los dos únicos pequeños inconvenientes de estas librerías "para novatos" (llamémoslas así) es que lógicamente no ofrecen toda la flexibilidad y capacidad que ofrece la librería SPI directamente, y que normalmente las librerías "para novatos" están enfocadas para un tipo determinado de dispositivo SPI muy concreto. Esto último implica que para cada dispositivo SPI deberemos de utilizar una librería "para novatos" diferente (cuando todos ellos en realidad de manejan por debajo con una única librería, la SPI). Además, corremos el riesgo de que para ese dispositivo en concreto que queremos usar nadie haya desarrollado su librería "para novatos" correspondiente aún. No obstante, a pesar de todo esto, para lograr códigos más comprensibles, en este libro

se ha optado por ver diversos ejemplos de manejo de dispositivos SPI haciendo uso de sus librerías particulares.

Librería Wire

Permite comunicar mediante el protocolo I^2C (también llamado TWI) la placa Arduino con dispositivos externos. Tal como ya se ha comentado varias veces, la comunicación se establece a través de los pines SDA (línea de datos, ubicada en pin analógico 4 en la placa Arduino UNO) y SCL (línea de reloj, ubicada en el pin analógico 5). Esta librería utiliza 7 bits para identificar el dispositivo (lo que da la posibilidad de distinguir hasta 128 dispositivos diferentes).

De todas formas, en este libro no estudiaremos el funcionamiento de esta librería, por los mismos motivos comentados en el apartado correspondiente a la librería SPI. Al igual que esta, la librería Wire es bastante compleja porque permite controlar hasta el mínimo detalle el proceso de comunicación entre dispositivos. Es por eso que en este libro se optará por utilizar librerías "para novatos" que ocultan los entresijos internos del proceso comunicativo y ofrecen al usuario una forma mucho más sencilla, clara y rápida de escribir sus sketches. Normalmente, cada dispositivo diferente utilizará su librería "para novatos" correspondiente, por lo que nos encontraremos con bastante variedad de librerías para elegir según el dispositivo que queramos usar, a pesar de que todas ellas se basen en una única librería base, la Wire.

Librería SoftwareSerial

Ya sabemos que gracias al chip TTL-UART de la placa UNO, esta es capaz de comunicarse a través de los pines 0 (RX) y 1 (TX) con dispositivos externos estableciendo una conexión serie. La librería SoftwareSerial lo que permite es que este tipo de comunicación se pueda establecer con dispositivos conectados a pines diferentes del 0 y del 1.

Por eso esta librería se llama así: porque consigue simular un chip TTL-UART "virtual" mediante software. Gracias a ella, es posible tener múltiples puertos serie con una velocidad de transferencia de hasta 115200 bits/s. Y al igual que el chip TTL-UART "real", cada puerto serie "simulado" dispone de un buffer de 64 bytes, de tal manera que el microcontrolador pueda acumular allí los datos recibidos mientras está trabajando en otras cosas.

No obstante, existen varias limitaciones: si se utilizan varios puertos, solamente uno puede recibir datos a la vez, y en determinadas placas no se pueden usar tampoco todos los pines digitales (en concreto, por ejemplo, en la placa Arduino

Leonardo para RX solamente se pueden utilizar los pines 8, 9, 10, 11, 14, 15 y 16). Estudiaremos el uso de esta librería en un apartado posterior de este capítulo.

Librerías Servo y Stepper

La librería Servo sirve para facilitar al programador de la placa Arduino el control de servomotores. Concretamente, permite manejar hasta 12 servomotores en la placa Arduino UNO (¡y hasta 48 en la Arduino Mega!). Solo hay que tener en cuenta el pequeño detalle de que el uso de esta librería en la placa Arduino UNO deshabilita la funcionalidad PWM de los pines 9 y 10 aunque no haya ningún servomotor conectado allí.

La librería Stepper, por su parte, sirve para controlar motores tipo "paso a paso" (en inglés, "steppers"), tanto de tipo unipolar como bipolar. Estudiaremos el uso de ambas librerías en un apartado posterior de este capítulo.

Librerías Keyboard y Mouse (solo para Arduino Leonardo y Due)

La librería Keyboard permite a las placas Arduino Leonardo, Arduino Micro y Arduino Due actuar como si fueran un teclado (simulando pulsaciones de teclas y enviándolas al computador conectado a ella), y la librería Mouse les permite actuar como si fueran un ratón (pudiendo controlar el movimiento del cursor en la pantalla del computador conectado a ella).

Respecto a la librería Keyboard, esta no es capaz de enviar todos los caracteres ASCII posibles –especialmente los no imprimibles–, pero sí que permite el uso de las llamadas teclas de modificación (los cursores, las teclas de función, etc.), las cuales suelen servir para cambiar el comportamiento de otra tecla pulsada simultáneamente. Para más información sobre la lista de teclas de modificación y su uso, se puede consultar http://arduino.cc/en/Reference/KeyboardModifiers. Estas librerías no las estudiaremos en este libro.

Librerías Audio, Scheduler y USBHost (solo para Arduino Due)

Las características hardware particulares de la placa Arduino Due permiten el uso de tres librerías extra que en el resto de placas Arduino no están disponibles: "Audio", "Scheduler" y "USBHost". Ninguna de estas librerías las estudiaremos.

La librería "Audio" permite a nuestros sketches reproducir ficheros de audio en formato WAV almacenados por ejemplo en una tarjeta SD. Esto es posible gracias a los dos conversores digital-analógicos que incorpora la placa Due, los cuales

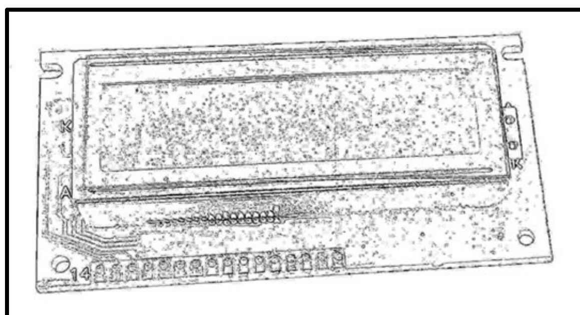
posibilitan tener dos salidas analógicas reales de 12 bits de resolución (es decir, de hasta 4096 niveles diferentes) con las que se puede generar audio de calidad.

La librería "Scheduler" permite a nuestros sketches ejecutar múltiples funciones al mismo tiempo. Esto permite realizar diferentes tareas a la vez sin que se interrumpan entre sí constantemente.

La librería "USBHost" permite a nuestros sketches actuar como "hosts" USB, de tal manera que se puedan conectar al conector USB mini-A de la placa Arduino Due diferentes periféricos USB, tales como teclados o ratones para así interactuar directamente con ella.

USO DE PANTALLAS

Las pantallas de cristal líquido (LCDs)



Las pantallas de cristal líquido (en inglés "Liquid Crystal Displays" –LCDs–) ofrecen una manera muy rápida y vistosa de mostrar mensajes. Las podemos clasificar en LCDs de caracteres y LCDs gráficas (estas últimas también llamadas GLCDs). Las primeras sirven para mostrar texto ASCII y se comercializan en diferentes tamaños (16x2, 20x4...) donde el primer número indica la cantidad de caracteres que caben en una fila, y el segundo número es el número de filas que caben en la pantalla. Las segundas sirven para mostrar, además de texto, dibujos e imágenes, y también se comercializan en diferentes tamaños, los cuales están definidos por la cantidad de píxeles que pueden mostrar (128x64, 128x128...). Las LCDs de caracteres, por su parte, pueden mostrar pequeños iconos de 5x7 píxeles o similar.

Las LCDs de caracteres más habituales son de 4-bit o 8-bit, dependiendo del número de cables (bits) que necesitan tener conectados al circuito para poder recibir o enviar datos. Ojo, solo se cuentan los cables que son estrictamente de transferencia de datos ya que en realidad una LCD necesita no solo 4 o 8 cables para funcionar, sino varios más (como los de alimentación, tierra, reseteado...). Otras características que

pueden tener (o no) las LCDs de caracteres son la posibilidad de iluminar el fondo de la pantalla (opción ideal para entornos con poca luz ambiental) o la posibilidad de utilizar varios colores de fondo (y no solamente el blanco/negro sobre azul/verde, que suele ser lo habitual), etc.

Cada modelo de LCD es diferente, por lo que es imprescindible consultar su datasheet concreto para poder distinguir los diferentes pines de conexión que ofrece y sus características generales. De todas formas, lo más habitual es que una LCD estándar ofrezca:

Un pin para recibir la alimentación (normalmente con los 5 V que proporciona la placa Arduino ya está bien, pero hay modelos que requieren 3,3 V, así que hay que vigilar esto) **y otro pin para conectar la pantalla a tierra**. Es conveniente conectar un divisor de tensión entre la fuente de alimentación y el pin de alimentación de la pantalla para evitar posibles daños. Para calcular el valor óptimo de esta resistencia, se deben consultar dos valores en el datasheet del LCD: la corriente máxima soportada para la luz de fondo y la caída de tensión causada por esta. Haciendo uso de la Ley de Ohm, si se resta dicha caída de tensión de los 5 V y se divide el resultado entre esa corriente máxima, obtendremos el valor de la resistencia (redondeando al alza) que necesitamos. Por ejemplo, si la corriente máxima es de 16 mA y la caída de tensión es de 3,5 V, la resistencia debería ser (5 – 3,5)/0,016 = 93,75 ohmios (o 100 ohmios redondeando a un valor estándar). Si no se puede consultar el datasheet, un valor seguro para usar son 220 ohmios, aunque un valor tan alto hará que la luz de fondo sea más tenue.

Un pin para regular el contraste de la pantalla. Este pin se debe conectar a la patilla central de un potenciómetro de nuestro circuito (el cual a su vez ha de tener sus patillas exteriores conectadas a la alimentación y tierra, respectivamente), de manera que regulando el potenciómetro podremos regular el contraste de la pantalla.

Tres pines de control generalmente marcados como "RS", "EN" y "RW", que se deberán conectar cada uno a un pin digital de la placa Arduino. El pin "RS" sirve para que el microcontrolador le diga a la LCD si quiere mostrar caracteres o si lo que quiere es enviar comandos de control (como cambiar la posición del cursor o borrar la pantalla por ejemplo). Concretamente, si por ese pin el LCD detecta una señal LOW, los datos recibidos serán tratados como comandos a ejecutar, y si detecta una señal HIGH, los datos recibidos

serán el texto a mostrar en la pantalla. El pin “EN” establece la línea “enable”, la cual sirve para advertir a la LCD que el microcontrolador le va a enviar datos (ya sean de control o para imprimir). Esta advertencia se produce cada vez que la señal recibida por ese pin cambia de HIGH a LOW. Finalmente, el pin “RW” sirve para definir si se desea enviar datos a la LCD (lo más común) o recibirlos de ella (muy poco común); si estamos en el primer caso, este pin deberá recibir una señal LOW y en el segundo caso deberá recibir una señal HIGH, por lo que, como normalmente no lo necesitaremos para nada, lo conectaremos casi siempre a tierra.

Varios pines (4 u 8, según si la LCD es de “4-bit” o “8-bit”) que se deberán conectar también cada uno a un pin digital de la placa Arduino. Se usan para establecer las líneas de comunicación en paralelo por donde se transfieren los datos y los comandos de control de la placa Arduino hacia el LCD. Hay que saber que una LCD de 8 bits puede funcionar perfectamente con solo cuatro cables de datos conectados (es decir, funcionando como una LCD de 4 bits), pero lo hará a una velocidad menor.

Dos pines exclusivos para el circuito de la luz de fondo (uno para recibir la alimentación y el otro pin para conectar a tierra). Si la pantalla no dispone de luz de fondo (también llamada de “retroalimentación”), estos pines o no existirán o no serán usados para nada.

Insisto que la lista anterior de “pines necesarios” es solo un “mínimo común” de entre todos los modelos de LCDs que hay en el mercado. Es absolutamente necesario, pues, consultar el datasheet del modelo que tengamos a mano para confirmar de cuántos pines consta ese LCD en particular, qué función realizan y qué ubicación tienen. Hay varios modelos, por ejemplo, que ofrecen la posibilidad de cambiar el color de la luz de fondo a uno cualquiera; esto es posible porque disponen internamente de tres LEDs (cada uno de un color primario –rojo, verde y azul–), los cuales deben ser conectados a pines de salida PWM de la placa Arduino mediante **tres pines más** ofrecidos por la pantalla, aparte de los ya descritos. Esta característica se llama RGB (de “Red-Green-Blue”).

Como guía para el lector, podemos nombrar como ejemplo de pantallas LCD estándar de 8-bits los productos siguientes de Adafruit: el nº 181 (de 2x16, caracteres blancos sobre fondo azul), el nº 198 (de 4x20, blanco sobre azul), el nº 399 (2x16, caracteres de colores RGB sobre fondo negro) o el nº 398 (2x16, caracteres negros sobre fondo RGB). Sparkfun por su parte también distribuye, entre otros modelos, el producto nº 255 (2x16, negro sobre verde), el nº 254 (4,x20, negro sobre verde), el nº

709 (2x16, blanco sobre negro), el nº 790 (2x16, amarillo sobre azul) o el nº 791 (2x16, rojo sobre negro). Todos ellos son compatibles con la librería "LiquidCrystal" oficial de Arduino.

La librería LiquidCrystal

Lo primero que debemos hacer para poder utilizar pantallas LCD compatibles con la librería oficial LiquidCrystal es declarar una variable global de tipo "LiquidCrystal", la cual representará dentro de nuestro sketch al objeto LCD que queremos controlar. La declaración se ha de realizar usando la siguiente sintaxis (suponiendo que llamamos "milcd" a dicha variable-objeto): *LiquidCrystal milcd(rs, rw, enable, d0, d1, d2, d3, d4, d5, d6, d7);* donde todos los parámetros especificados entre paréntesis en realidad son valores numéricos que representan:

rs : nº del pin de la placa Arduino conectado al pin "RS" de la LCD.
rw: nº del pin de la placa Arduino conectado al pin "RW" de la LCD. Opcional.
enable: nº del pin de la placa Arduino conectado al pin "ENABLE" de la LCD.
d0... hasta **d7**: números de los pines de la placa Arduino conectados a los pines de datos correspondientes de la LCD. Los parámetros *d0, d1, d2* y *d3* son opcionales: si se omiten, la LCD será controlada usando solo cuatro líneas (d4, d5, d6, d7) en vez de 8.

Por ejemplo, una posible declaración para una LCD de 4 bits sin uso del pin "rw" podría ser: `LiquidCrystal milcd(12, 11, 5, 4, 3, 2);.`

Una vez creado el objeto *milcd* con la línea anterior, lo primero que debemos hacer es establecer el tamaño de la pantalla para poder trabajar con ella. Esto se hace mediante la siguiente función:

milcd.begin(): especifica las dimensiones (columnas y filas) de la pantalla. Tiene dos parámetros: el primero es el número de columnas que tiene la pantalla y el segundo es el número de filas. No tiene valor de retorno. Esta función ha de ser ejecutada antes de poder empezar a imprimir ningún carácter en ella.

A partir de aquí, ya podremos escribir caracteres en la pantalla con alguna de las tres funciones siguientes:

milcd.write(): escribe un carácter en la pantalla. Como único parámetro tiene ese carácter, el cual se puede especificar explícitamente entre comillas simples o bien a través de una variable de tipo "byte" (cuyo valor puede haber sido obtenido de un sensor, o leído del canal serie, etc.). Su dato de

retorno es de tipo “byte” y vale el número de bytes escritos (por tanto, si todo es correcto, valdrá 1), aunque no es obligatorio utilizarlo.

milcd.print(): escribe un dato (de cualquier tipo) en la pantalla. Como primer parámetro tiene ese dato, que puede ser tanto un carácter de tipo “char” como una cadena de caracteres, pero también puede ser numérico entero (“int”, “long”, etc.). Opcionalmente, se puede especificar un segundo parámetro que indica, en el caso de que el dato a escribir sea entero, el formato con el que se verá: puede valer la constante predefinida BIN (que indicará que el número se visualizará en formato binario), HEX (en hexadecimal) y DEC (en decimal, que es su valor por defecto). Su dato de retorno es de tipo “byte” y vale el número de bytes escritos, aunque no es obligatorio utilizarlo.

milcd.createChar(): crea un carácter totalmente personalizado. Se pueden crear hasta ocho caracteres diferentes de 5x8 píxeles cada uno. La apariencia de cada carácter es especificada por un array de 8 bytes. Cada uno de estos bytes representa una fila de píxeles. Cada fila de píxeles se dibuja dependiendo de los valores de los últimos cinco bits (los de más a la derecha) de su byte correspondiente: si el bit vale 1, se pintará el píxel pertinente y si vale 0, no. No obstante, esta función no escribe el carácter en la pantalla, tan solo lo crea; para poderlo escribir, deberemos utilizar *milcd.write()*. Esta función tiene dos parámetros: el primero es la identificación del carácter a crear (puede ser un número entre 0 y 7) y el segundo es el array con el “dibujo” de los píxeles. Para transformar el dibujo que queremos en los valores adecuados del array, podemos usar la ayuda interactiva que ofrece la página http://www.quinapalus.com/hd44780udg.html.

Ejemplo 5.1: Veamos un ejemplo de uso de las funciones anteriores. Suponiendo que tenemos ya conectado el pin *rs* del LCD al pin nº 12 de la placa Arduino, el pin *enable* al pin número 11 y los cuatro pines de datos *d4, d5, d6* y *d7* a los pines 5, 4, 3 y 2, (además de tener conectados también como mínimo los pines de alimentación y tierra del LCD), el siguiente código mostraría un icono sonriente en la pantalla:

```
#include <LiquidCrystal.h>
LiquidCrystal milcd(12, 11, 5, 4, 3, 2);
/*En decimal los valores de la inicialización
 del array “smiley” serían: 0,17,0,0,17,14,0 */
byte smiley[8] = {
 B00000, B10001, B00000, B00000, B10001, B01110, B00000,
};
```



```
void setup() {
        milcd.createChar(0, smiley);
        //La pantalla es de 2 filas de 16 caracteres
        milcd.begin(16, 2);
        milcd.write(0);
}
void loop() {}
```

A partir de aquí, podemos manipular la pantalla de diferentes formas con la ayuda de las instrucciones siguientes:

milcd.clear(): borra la pantalla y posiciona el cursor en su esquina superior izquierda para escribir a partir de allí el próximo texto. No tiene ni parámetros ni valor de retorno.

milcd.home(): posiciona el cursor en la esquina superior izquierda de la pantalla para escribir a partir de allí el próximo texto. Si además se quiere borrar la pantalla, entonces se ha de utilizar *milcd.clear()*. No tiene ni parámetros ni valor de retorno.

milcd.setCursor(): posiciona el cursor en la columna y fila especificadas como parámetros para escribir a partir de allí el próximo texto. Su primer parámetro es la columna en la que se quiere situar el cursor (la primera es la número 0) y su segundo parámetro es la fila (la primera es la número 0 también). No tiene valor de retorno.

milcd.cursor(): muestra el cursor en forma de línea de subrayado en la posición donde el próximo carácter será escrito. No tiene ni parámetros ni valor de retorno. También existe la función **milcd.noCursor()**, la cual oculta el cursor (y tampoco tiene ni parámetros ni valor de retorno).

milcd.blink(): muestra el cursor parpadeando (suponiendo que este sea visible). No tiene ni parámetros ni valor de retorno. También existe la función **milcd.noBlink()**, la cual evita el parpadeo del cursor (y tampoco tiene ni parámetros ni valor de retorno).

milcd.display(): enciende la pantalla (después de haberla apagado mediante **milcd.noDisplay()**). La función *milcd.noDisplay()* apaga la pantalla pero no pierde el texto actual, por lo que si se ejecuta *milcd.display()* se volverá a restaurar el texto (y cursor) que se estaba mostrando previamente. Ninguna de las dos funciones tiene ni parámetros ni valor de retorno.

milcd.scrollDisplayLeft(): desplaza el contenido de la pantalla (texto y cursor) un espacio hacia la izquierda. No tiene ni parámetros ni valor de retorno. También existe la función **milcd.scrollDisplayRight()**, que desplaza el contenido un espacio hacia la derecha (y que tampoco tiene ni parámetros ni valor de retorno).

milcd.autoscroll(): activa el desplazamiento automático del contenido mostrado en la pantalla. Esto provoca que cada carácter sea "empujado" en un espacio por el siguiente, en el sentido marcado por las funciones *milcd.leftToRight()* o *milcdrightToLeft()* –ver párrafo siguiente–. El efecto visual es que cada carácter nuevo aparecerá en la misma posición dentro de la pantalla. No tiene parámetros ni valor de retorno. También existe la función **milcd.noAutoscroll()**, la cual evita este comportamiento (y tampoco tiene ni parámetros ni valor de retorno).

milcd.leftToRight(): establece el sentido de escritura del texto en la pantalla. Por defecto ya es de izquierda a derecha (es decir, que los caracteres que se vayan imprimiendo lo harán en ese sentido), por lo que no haría falta de entrada ejecutar esta función, a no ser que previamente se hubiera ejecutado **milcd.rightToLeft()**, la cual establece el sentido de escritura de derecha a izquierda. Ninguna de estas dos funciones afectará en cualquier caso al texto previamente escrito en la pantalla antes de su ejecución, y no tienen ni parámetros ni valor de retorno.

Ejemplo 5.2: Veamos un ejemplo de uso de algunas de las funciones anteriores, suponiendo las mismas conexiones entre placa Arduino y LCD:

```
#include <LiquidCrystal.h>
LiquidCrystal milcd(12, 11, 5, 4, 3, 2);
void setup() {
      milcd.begin(16,2);
      milcd.print("Hola");
}
void loop() {
      // Me sitúo en el primer carácter de la segunda fila
      milcd.setCursor(0,1);
      // Escribo los segundos desde el inicio del sketch
      milcd.print(millis()/1000);
}
```

Ejemplo 5.3: Otro ejemplo muy sencillo es el siguiente, que muestra en el LCD aquello que se escribe en el "Serial monitor" en tiempo real (suponiendo que nuestra placa Arduino la tenemos conectada a un computador vía USB).

```
#include <LiquidCrystal.h>
LiquidCrystal milcd(12, 11, 5, 4, 3, 2);
int luzFondo = 13; //Defino el pin para la luz de fondo
void setup() {
     pinMode(luzFondo, OUTPUT);
     digitalWrite(luzFondo, HIGH); //Activo la luz de fondo
     lcd.begin(20,4);
     Serial.begin(9600);
}
void loop() {
     if (Serial.available()>0) {
     //Me espero un poco para que llegue el mensaje entero
            delay(100);
            lcd.clear();
            //Leo todos los caracteres disponibles del buffer
            while (Serial.available() > 0) {
                 //Muestro cada carácter en el LCD
                 lcd.write(Serial.read());
            }
     }
}
```

Ejemplo 5.4: Otro ejemplo de código es el siguiente, el cual muestra en el LCD una secuencia de caracteres que van posicionándose cada vez más a la derecha, a modo de barra de progreso animada.

```
#include <LiquidCrystal.h>
LiquidCrystal milcd(12, 11, 5, 4, 3, 2);
/*El código muestra en una posición tres caracteres personalizados, uno tras otro, y entonces paso a la siguiente posición y vuelvo a repetir la secuencia */
byte a[8] = {B00000,B00000,B00000,B00100,B00100,B00000,
B00000,B00000};
byte b[8] = {B00000,B00000,B10001,B10001,B10001,B10001,
B00000,B00000};
byte c[8] = {B11111,B10001,B10001,B10001,B10001,B10001,
B10001,B11111};
void setup() {
      milcd.createChar(0,a);
```

```
      milcd.createChar(1,b);
      milcd.createChar(2,c);
      milcd.begin(16,2);
      milcd.clear();
}
void loop() {
       int i;
       for (int i=0; i<16; i++){
             milcd.setCursor(i,0); milcd.write(0); delay(250);
             milcd.setCursor(i,0); milcd.write(1); delay(250);
             milcd.setCursor(i,0); milcd.write(2); delay(250);
             milcd.setCursor(i,0); milcd.write(1); delay(250);
             milcd.setCursor(i,0); milcd.write(0); delay(250);
       }
       delay(1000);
       milcd.clear();
}
```

Si se quieren mostrar caracteres de un tamaño mayor al estándar utilizado en los LCDs compatibles con el chip HD4478, se puede utilizar las librerías "Phi_big_font" y "Phi_super_font", ambas descargables de http://code.google.com/p/phi-big-font. La primera permite mostrar caracteres de un tamaño hasta 6 veces mayor y la segunda de hasta 40 veces mayor. La primera requiere el uso de dos filas y cuatro columnas para mostrar un carácter, con una columna en blanco para la separación entre uno y otro; esto significa que en una pantalla de 16x2 caben 4 caracteres y en una de 20x4 caben 5 en 2 filas. La segunda requiere el uso de cuatro filas y 5 columnas para mostrar un carácter, con una columna en blanco para la separación entre uno y otro; esto significa que en una pantalla de 20x4 caben 4 caracteres. Ambas librerías admiten animaciones.

Módulos LCD de tipo I^{2}C, Serie o SPI

Las pantallas LCD comunes tienen el inconveniente de requerir muchos cables para conectarse al circuito. Como consecuencia, en nuestra placa Arduino nos pueden quedar pocos pines disponibles para usarlos en otras cosas. Una solución a este inconveniente es el empleo de pantallas LCD que utilizan un sistema de comunicación con la placa Arduino diferente del ya comentado, como pueden ser los protocolos I^{2}C (el cual solo utiliza las líneas SDA y SCL), o el serie (usando solo las líneas RX y TX), principalmente.

En realidad, estas pantallas LCD son exactamente las mismas que las listadas en el apartado anterior, pero incorporan una pequeña plaquita en su dorso (llamada

"backpack") que incluye un pequeño microcontrolador encargado de realizar la traducción del protocolo adecuado (I^2C, Serie, etc.) a las señales "estándares" que la pantalla LCD entiende.

Por ejemplo, DFRobot distribuye un conjunto llamado "I^2C LCD1602 module" formado por una pantalla alfanumérica de 2x16 caracteres más un backpack y otro llamado "I^2C LCD2004 module" formado por una pantalla de 4x20 caracteres más el mismo backpack. Estos conjuntos pantalla+backpack tan solo requieren cuatro cables para ser controlados totalmente desde la placa Arduino: alimentación de 5 V, tierra, cable SDA y cable SCL. Además, disponen de un potenciómetro para ajustar el contraste y se programan mediante una librería propia (descargable de la página web de ambos productos) llamada "LiquidCrystal_I2C", prácticamente idéntica a la librería oficial de Arduino.

Otro producto muy parecido al anterior es el "TWILCD" de Akafugu: al igual que las pantallas de DFRobot, esta de Akafugu también se conecta a la placa mediante alimentación, tierra, SDA y SCL, y también se programa mediante una librería propia (descargable en este caso de https://github.com/akafugu/twilcd) prácticamente idéntica a la librería oficial de Arduino. No obstante, en vez de utilizar un potenciómetro para regular el contraste, este se puede controlar desde software. Otra diferencia es que el backpack se puede adquirir junto con la pantalla alfanumérica pero también por separado (para utilizar una pantalla que tengamos ya, siempre que sea compatible con el chip Hitachi HD44780 o similar y sus dimensiones sean 1x16 o 2x16 o 1x20 o 2x20). Si queremos manejar LCDs de tipo RGB o bien de un tamaño mayor (como 4x40, por ejemplo), en Akafugu ofrecen (con pantalla opcional) el backpack llamado "TWILCD 40x2/40x4/RGB".

Adafruit también distribuye un backpack de tipo I^2C (producto nº 292), pero no ofrece la posibilidad de adquirirlo junto con una pantalla ya preensamblada, sino que es necesario soldarlo manualmente con alguna de las pantallas LCD alfanuméricas existentes en su catálogo. En todo caso, para poderlo controlar es necesario utilizar una librería propia (prácticamente idéntica a la oficial de Arduino) descargable de http://github.com/adafruit/LiquidCrystal.

Otro backpack I^2C (con pantalla LCD opcionalmente ya preensamblada) es el "SR LCD" de http://www.electrofunltd.com, programable mediante una librería propia descargable de su web, también prácticamente idéntica a la oficial.

Por otro lado, también existen pantallas LCD que, gracias a un chip TTL-UART que llevan incorporado, implementan el protocolo serie. De esta manera, con solo tres cables (alimentación, tierra y la conexión del pin RX de la pantalla al pin TX de la

placa Arduino) ya podemos controlarlas. Tan solo es cuestión de enviarles mediante *Serial.write()* algunos comandos propios específicos para cada acción que deseemos realizar (activar luz de fondo, limpiar pantalla, posicionar el cursor, etc.), y mediante *Serial.print()* el texto a imprimir. Para conocer estos comandos se ha de consultar el datasheet respectivo. Sparkfun, por ejemplo, distribuye varios modelos de este tipo, como los productos nº 9393, nº 9394, nº 9395, nº 9396 o nº 9568, entre otros.

Ejemplo 5.5: El siguiente código es un ejemplo de prueba para cualquiera de las pantallas LCD serie de Sparkfun.

```
/*El texto a mostrar se envía por el canal serie directamente. Los
comandos de control se especifican con dos bytes: el primero es un
comando genérico que "advierte" al LCD del tipo de acción a realizar,
y el segundo es el comando que realiza esa acción. Es conveniente
utilizar Serial.write() debido a que los comandos son en realidad
valores hexadecimales (con Serial.print() se podrían confundir con su
representación ASCII). Los delay() son necesarios porque si los
comandos se envían demasiado seguidos, el LCD puede no recibirlos
correctamente.*/
void setup(){
  Serial.begin(9600); //El LCD funciona por defecto a esa velocidad
  luzfondoOn();  irLineaUno();  Serial.print("Hola");
  irLineaDos();  Serial.print("Bienvenido");   delay(100);
}
void loop(){}
//Enciende la luz de fondo
void luzfondoOn(){
    //Comando de control para gestionar la luz de fondo
    Serial.write(0x7C);
    //Comando que establece la máxima cantidad de luz posible
    Serial.write(157);  delay(5);
}
//Apaga la luz de fondo
void luzfondoOff(){
    Serial.write(0x7C);
    Serial.write(128);   //128 es el valor mínimo (luz apagada)
    delay(5);
}
//Coloca el cursor en el carácter 0 de la primera línea
void irLineaUno(){
   //Comando de control para gestionar la posición del cursor
   Serial.write(0xFE);
   //Comando que cambia efectivamente la posición a la indicada
```



```
   Serial.write(128);  delay(5);
}
//Coloca el cursor en el carácter 0 de la segunda línea
void irLineaDos(){
   Serial.write(0xFE); Serial.write(192); delay(5);
}
/*Coloca el cursor en cualquier posición
(la primera fila es la n° 0 y la primera columna también)*/
void irPosicion(int fila, int col) {
  //Comando de control para gestionar la posición del cursor
  Serial.write(0xFE);
  Serial.write((col + fila*64 + 128));  //Posición deseada
  delay(5);
}
//Otra función interesante. Para saber más, consultar el datasheet
void borrarLCD(){
   //Comando de control para gestionar el borrado del LCD
   Serial.write(0xFE);
   //El comando propiamente dicho que borra la pantalla
   Serial.write(0x01);   delay(5);
}
```

De todas formas, existe una librería llamada “SerLCD” que permite utilizar los LCDs serie de Sparkfun mediante las mismas funciones que las de la librería LiquidCrystal oficial de Arduino, facilitando enormemente su gestión. Se puede descargar de aquí: http://arduino.cc/playground/Code/SerLCD.

Otro backpack serie (con pantalla LCD opcionalmente ya preensamblada) es el “LCD117 Serial LCD Kit” de Modern Device; en la página del producto se ofrece la referencia completa de comandos que acepta vía *Serial.write()*, junto con varios códigos Arduino de ejemplo y, además, un software específico para el diseño de caracteres personalizados. Otro producto similar (con la pantalla LCD ya preensamblada de fábrica) es el nº 27977 de Parallax; en su página se ofrece toda la documentación necesaria para aprender su uso, y varios códigos Arduino de ejemplo.

Por otro lado, existen LCDs que además de ofrecer conectividad serie como los modelos de Sparkfun, son capaces de comunicarse directamente con un computador a través de un cable USB para recibir los comandos de control o el texto a mostrar mediante un software de tipo “terminal serie”. Un ejemplo de estos LCDs son los productos nº 782 o nº 784 de Adafruit (ambos con capacidad RGB).

Finalmente, otro módulo LCD digno de destacar es el “3-Wire Serial LCD module” de DFRobot, que, tal como su nombre indica, también solo requiere el uso de tres cables. Incorpora una pantalla gráfica de 128x64 píxeles y es programable mediante una librería particular suya derivada de la “SPI” oficial, descargable de la web del producto.

Shields que incorporan LCDs

Aparte de los módulos LCD ya mencionados o similares, podemos adquirir para nuestros proyectos shields que integran una LCD y que tienen la ventaja de ser directamente empotrables sobre nuestra placa Arduino. De esta manera, tendremos un circuito más compacto. Además, estos shields no incorporan solo una pantalla sino que además suelen venir con un conjunto de pulsadores (como mínimo cuatro para hacer la función de cursores por la pantalla y poder navegar por los menús, además de otro botón más de control, para seleccionar la opción elegida) que permiten añadir interactividad a nuestros proyectos y poder así enviar órdenes a la placa Arduino sin necesitar ningún computador externo. A continuación, se nombran algunas de las diferentes alternativas que tenemos.

DFRobot fabrica su “LCD keypad shield”, que incorpora una pantalla LCD alfanumérica de 2x16, de fondo azul y caracteres blancos, con soporte para ajuste de contraste y activación o desactivación de la luz de fondo. El shield está controlado por el chip HD44780 de Hitachi, por lo que la pantalla se puede programar mediante la librería de programación oficial del lenguaje Arduino, aunque también se puede utilizar la librería “LCD4bit_mod”, descargable de su web. Además, este shield tiene 5 con botones de control (arriba, abajo, derecha e izquierda y selección) y un botón de reinicio; para saber qué botones se están pulsando en un momento dado, nuestro programa debe ir leyendo en cada “loop” la señal recibida por el pin analógico nº 0: según el valor numérico (en diferentes rangos de 0 a 1023) que tenga esa señal, se podrá saber qué botón ha sido pulsado y actuar en consecuencia. Por otro lado, los pines del shield del 4 al 10 son usados para interactuar internamente con la pantalla, por lo que no se pueden usar para otra cosa; para compensar esta pérdida de pines, este shield aporta varios pines analógicos de entrada extra.

Otros shields prácticamente idénticos al anterior (ya que utilizan el mismo chip de Hitachi, también tienen pantallas de 2x16 y también incorporan botones de dirección y control gestionables a través de una entrada analógica) son el “LCD keypad shield” de IteadStudio, y el “LCD & Keypad shield” de Freetronics programables ambas mediante la librería oficial de Arduino. Para gestionar los pulsadores, utilizan la misma técnica de leer en cada “loop” la señal recibida por el

pin analógico nº 0, comprobar su valor numérico y actuar en consecuencia. Aunque en las respectivas páginas del producto se ofrecen códigos de ejemplo mostrando esta técnica, si se desea controlar las pulsaciones de los botones de una forma más sencilla, tanto el shield de IteadStudio como el de Freetronics se pueden programar mediante la librería “LCD” (http://rweather.github.com/arduinolibs/classLCD.html).

El “RGB LCD shield kit” de Adafruit es muy parecido a los anteriores shields pero incluye la novedad de que su pantalla 2x16 permite el cambio del color de fondo; es por eso que, aunque se puede programar mediante la librería oficial de Arduino, Adafruit ofrece una librería optimizada para este shield en concreto llamada “Adafruit RGB”, descargable desde https://github.com/adafruit/Adafruit-RGB-LCD-Shield-Library .

No obstante, esta librería no nos será útil si no tenemos instalada otra: la librería “Adafruit GFX”, disponible en https://github.com/adafruit/Adafruit-GFX-Library. Esta última es realmente la que utilizaríamos para pintar los píxeles (y para dibujar las líneas, círculos, e incluso imágenes a todo color si la pantalla fuera de tipo gráfico), ya que la “Adafruit RGB” en realidad tan solo sirve para configurar la comunicación interna entre shield y placa Arduino. De hecho, “Adafruit GFX” funciona junto con muchas otras librerías (listadas en http://learn.adafruit.com/adafruit-gfx-graphics-library) todas ellas desarrolladas por Adafruit para diferentes modelos concretos de pantallas (GLCDs, OLEDs, TFTs, etc.), siendo la “Adafruit RGB” solo una más. Esta separación entre librerías específicas para un determinado hardware y una única librería gráfica común permite disponer, para diferentes tipos de pantallas, de una sola sintaxis de programación. De esta manera, los sketches Arduino se pueden adaptar fácilmente con los mínimos cambios. Las funciones de la librería “Adafruit GFX” son bastante intuitivas (son del tipo *milcd.drawLine(), milcd.drawCircle()* o *milcd.fillTriangle()*, por ejemplo) pero si se quiere profundizar en su estudio, recomiendo consultar el enlace anterior.

También podemos encontrar útil el shield “DeuLigne” de Snootlab, el cual, aunque aparentemente sea muy parecido a los anteriores (chip de Hitachi, pantalla 2x16, joystick de 5 posiciones leídas por una entrada analógica) utiliza el protocolo I^2C en vez del paralelo de 4 o 8 bits. Esto significa que solo necesita dos pines analógicos (los nº 4 y nº 5) para comunicarse con la placa Arduino, y que por tanto, libera muchos pines para otros usos. Es necesario, no obstante, utilizar una librería propia, descargable de https://github.com/Snootlab/Deuligne. Como ventaja, tenemos que esta librería es capaz no solo de gestionar la pantalla sino que también de una forma integral maneja las pulsaciones de los botones.

Finalmente, merece la pena destacar el "Phi-2 shield" (http://liudr.wordpress.com/shields/phi-2-shield), el cual se distribuye en dos versiones: con pantalla de 16x2 o 20x4. Al igual que los anteriores, este shield ofrece un conjunto de pulsadores para interactuar con la pantalla, pero la gran novedad es que para un control exhaustivo de estos disponemos de una estupenda librería llamada "Phi_prompt" (http://code.google.com/p/phi-prompt-user-interface-library), la cual incluye funciones para generar menús interactivos, para manejar áreas de texto desplazables vertical y horizontalmente o para gestionar entradas de números y caracteres para distintas disposiciones de botones. Gracias a ella, la introducción o selección de opciones en nuestro programa se realiza de forma muy simple. Para funcionar, "Phi_prompt" necesita la instalación de otra librería suplementaria, descargable desde el mismo sitio, la llamada "Phi_interfaces", la cual aporta una capa abstracción común y homogénea para diferentes tipos de entradas: pulsadores digitales, pulsadores analógicos, keypads, etc.

Por otro lado, merece destacar la existencia de una librería (independiente del modelo concreto de shield utilizado) que permite programar de forma muy sencilla la interactividad con menús (es decir, la navegación, la selección de opciones, etc.) en LCDs compatibles con el chip HD4478. Esta librería se llama "Menwiz" y está disponible en https://github.com/brunialti/MENWIZ . Se basa a su vez en otra librería más básica, la "New LiquidCrystal" (https://bitbucket.org/fmalpartida/new-liquidcrystal), la cual no es más que una mejora de la librería oficial de Arduino que permite obtener una mayor velocidad de funcionamiento y también una mayor versatilidad al admitir otros protocolos de comunicación con LCDs (I^2C, serie, etc.).

Shields y módulos que incorporan GLCDs

Sparkfun ofrece su "Color LCD Shield" –producto nº 9363– , que incorpora la pantalla Nokia 6100 (de 128x128 píxeles) y tres pulsadores de control. Los colores que puede mostrar la pantalla, además del blanco y del negro, son nueve: rojo, verde, azul, cian, magenta, amarillo, marrón, naranja y rosa. Se comunica con la placa Arduino mediante SPI a través de los pines digitales nº 8, 9 ,11 y 13 y usa los pines digitales nº 3, 4 y 5 para los tres pulsadores. Para programar sketches que hagan uso de este shield recomiendo usar la librería "ColorLCDShield", descargable de https://github.com/jimblom/ColorLCDShield. En el archivo "README" que viene incluido junto con la librería descargada se puede consultar un resumen de las instrucciones que podemos utilizar, las cuales nos permiten dibujar círculos, rectángulos, líneas y cadenas de caracteres de diferentes colores y tamaños de una forma muy sencilla. También podemos consultar los distintos sketches de ejemplo de uso que vienen incluidos dentro del paquete descargado.

Otro shield es el llamado "Graphic LCD4884 shield", fabricado por DFRobot. Incluye una pantalla gráfica monocromática de 48x84 píxeles (la Nokia 5110/3310) y, en vez de incluir pulsadores de control, incluye un joystick de 5 posiciones conectado a un único pin de entrada analógico (el nº 0), por lo que se tienen más pines-hembra libres disponibles. También se comunica con la placa Arduino mediante SPI y es programable mediante la librería "LCD4884", descargable de su web.

IteadStudio por su parte distribuye los shields llamados "IBridge" y "IBridge Lite", ambos con la misma pantalla Nokia 5100/3310 pero con un keypad de 9 y 12 botones, respectivamente. Se pueden controlar mediante una librería básica descargable de la página web de ambos productos.

Por otro lado, si preferimos utilizar módulos independientes en vez de shields, Sparkfun ofrece en forma de plaquita breakout (producto nº 11062) la misma pantalla que viene en su "Color LCD shield", la Nokia 6100. Esta plaquita incorpora además dos pulsadores y se comunica utilizando un protocolo SPI de tres cables (gracias a que las líneas MOSI y MISO se unifican en un solo canal llamado DIO, de tipo half-duplex). La conexión más simple que podemos realizar entre el módulo y la placa Arduino se muestra en la tabla siguiente:

Placa breakout	Placa Arduino
S1 (pulsador 1)	-
S2 (pulsador 2)	-
CS	Pin digital nº 3
SCK	Pin digital nº 4
DIO	Pin digital nº 5
RESET	Pin digital nº 6
STAT2	-
STAT1	-
STAT0	-
GND	GND
3.3V	3V3
VBATT	5V

Si se desea utilizar los dos pulsadores, se deben conectar a sendos pines digitales de entrada de la placa Arduino que queden libres. Los conectores "STAT2", "STAT1" y "STAT0" sirven para conectar, respectivamente, tres LEDs incluidos en la placa, de color verde, azul y rojo. El pin "3,3 V" de la placa breakout sirve para alimentar su circuitería interna , y el pin "VBATT" sirve para alimentar la pantalla propiamente dicha (aumentando internamente el voltaje recibido hasta los 7 V

necesarios). Para programar sketches que hagan uso de este módulo, recomiendo utilizar la librería https://github.com/TCWORLD/gLCD-Library.

Otra pantalla GLCD que viene en forma de módulo es el producto nº 710 de Sparkfun, la cual tiene unas dimensiones de 128x64 e incorpora el chip controlador KS0108 de Samsung. Este módulo posee 20 pines de conexión, los cuales se han de conectar (del nº 1 al nº 20) a los siguientes pines-hembra de la placa Arduino: 5 V, GND, patilla central de un potenciómetro de 10 KΩ regulador de contraste, D8, D9, D10, D11, D4, D5, D6, D7, A0, A1, RST, A2, A3, A4, una patilla lateral del potenciómetro (la otra va a tierra), 5V y GND. Para programar sketches que hagan uso de este módulo, se puede usar la librería http://code.google.com/p/glcd-arduino, que de hecho es compatible con todos los módulos que incorporen el chip KS0108. Con ella podremos dibujar rectángulos, círculos, líneas, puntos… y cadenas de caracteres de múltiples colores y tamaños. Esta librería es muy completa y versátil y está muy bien documentada: dentro de la carpeta “doc” ubicada en el interior del contenido de esta podemos encontrar un extenso manual en pdf que explica claramente la configuración y uso básico de los LCDs que pueden ser usados con ella y también podemos disponer de su referencia detallada en formato html. También disponemos de sketches de ejemplo como muestra de lo que es capaz de realizar.

Por otro lado, Sparkfun también ofrece su producto nº 8799 (GLCD monocromática de 160x128) y su producto 8884 (GLCD monocromática de 160x128), ambos gestionados por el chip T6963C. Pero lo que tienen de especial es que acoplándoles un módulo llamado “Graphic Serial LCD Back Pack” (producto nº 9352) se puede establecer comunicación serie con ellas. Esto quiere decir que tan solo necesitaremos un cable que conecte el pin TX de nuestra placa Arduino con el pin RX del backpack para poderles enviar comandos mediante *Serial.write()* que controlen la visualización y que impriman los textos y dibujos deseados (comandos que debemos consultar en el datasheet del producto).

Una pantalla GLCD (de 128x64) que ya incorpora el backpack de fábrica (y que, por tanto, solo requiere tres cables para funcionar: alimentación, tierra y datos serie) es el producto nº 9351 de Sparkfun. Como es habitual en estos casos, mediante *Serial.write()* podemos enviar los comandos adecuados (consultables en la referencia del producto) para poder dibujar en ubicaciones exactas y a varios tamaños tanto píxeles individuales como líneas, círculos, rectángulos y textos, para poder cambiar los colores de fondo y primer plano, para poder cambiar el brillo, para poder borrar zonas concretas de la pantalla, etc. Si lo deseamos, en vez de escribir estos comandos hexadecimales directamente, para este modelo de pantalla podemos utilizar una sencilla librería que nos facilitará la vida, descargable de http://sourceforge.net/projects/serialglcdlib.

Gravitech ofrece, por su parte, el producto "132x132 Serial Color Graphic LCD": una plaquita breakout con la pantalla Nokia 6610 (la cual es idéntica a la Nokia 6100 ya conocida excepto en sus dimensiones algo mayores). También utiliza el protocolo SPI de tres cables para comunicarse con la placa Arduino, así:

Placa breakout	Placa Arduino
VBL (nº 1)	3V3 o 5V
VCC (nº 2)	3V3 o 5V
GND (nº 3)	GND
RESET (nº 4)	Pin digital nº 2
SDATA (nº 5)	Pin digital nº 3
SCLK (nº 6)	Pin digital nº 4
CS (nº 7)	Pin digital nº 5
BL (nº 8)	Pin digital nº 6

donde "VBL" es el pin por donde la placa breakout recibe la alimentación necesaria para la pantalla (la cual internamente se eleva hasta los 7 V), "VCC" es por donde recibe la alimentación necesaria para su circuitería interna, "SDATA" es el canal half-duplex de comunicación, y "BL" sirve para enviar una señal HIGH o LOW según deseemos activar la iluminación de la pantalla o no. Este producto no viene acompañado de ninguna librería Arduino específica, pero Gravitech nos proporciona códigos de ejemplo fácilmente adaptables a nuestros propósitos particulares.

Si, a pesar de todo, disponemos de un módulo o shield que incorpora un LCD o GLCD no compatible con ninguno de los listados en estas páginas, siempre se puede consultar un listado muy extenso de librerías desarrolladas por la comunidad Arduino para diferentes modelos de pantallas en http://arduino.cc/playground/Code/LCD.

Shields que incorporan pantallas OLED de 4DSystems

Aparte de las pantallas de cristal líquido, existen otros tipos de pantalla construidas con diferentes tecnologías y materiales. Uno de esos tipos son las pantallas OLED, las cuales están compuestas por un tipo especial de LEDs (de ahí su nombre), son mucho más delgadas y flexibles, ofrecen más contraste y brillo (pueden utilizarse incluso a plena luz del sol), tienen mayor ángulo de visión y realizan un menor consumo (por, entre otras cosas, no necesitar luz de fondo). Sin embargo, la relativamente rápida degradación de los materiales con los que se fabrican ha limitado de momento su uso (aunque se está investigando para dar solución a estos problemas).

Las pantallas OLED permiten dibujar gráficos a todo color: desde simples píxeles, rectángulos o círculos hasta fotografías y vídeo. Además, todos los modelos suelen disponer de un zócalo para tarjetas de memoria microSD que permite almacenar tanto datos como imágenes, animaciones o vídeos que luego se reproducen directamente en la pantalla.

El fabricante 4Dsystems ofrece un conjunto de shields con pantalla OLED incorporada compatibles con Arduino. Concretamente, son cuatro modelos que se diferencian básicamente en la resolución y tamaño de su pantalla: el "4Display-Shield-96" (con una pantalla PMOLED de 96x64 píxeles y 20x14 mm), el "4Display-Shield-128" (con una pantalla PMOLED de 128x128 píxeles y 27x27 mm), el "4Display-Shield-144" (con pantalla LCD-TFT de 128x128 píxeles y 25,5x26,5 mm) y el "4Display-Shield-160" (con pantalla PMOLED de 160x128 píxeles y 33,6x27 mm). Cada una de estas pantallas incorpora de forma integrada un chip controlador diferente (el uOLED-96-G1, el uOLED-128-G1, el uLED-144 y el uOLED-160-G1, respectivamente), pero todas ellas pueden mostrar la misma cantidad de colores (más de 65000). Por lo demás, todas las placas disponen de un pequeño joystick (conectado a los pines digitales nº 2, 3, 4, 5 y 6 del Arduino) que puede ser usado también como pulsador, de un zócalo microSD con soporte para tarjetas SD y SDHC y se alimentan con 5 V.

La comunicación entre estos shields y la placa Arduino se puede hacer de dos maneras según el firmware que tenga pregrabado de fábrica el chip gráfico GOLDELOX: si este se adquiere preconfigurado en modo SGC, la comunicación entre shield y placa se realiza a través del canal serie (pines RX y TX). En este modo de funcionamiento, podremos enviar un rico conjunto de comandos hexadecimales predefinidos que el chip en modo SGC será capaz de recibir y traducir a líneas, círculos, texto, imágenes o vídeo.

De todas formas, aunque al shield se le pueden enviar directamente estos comandos mediante el uso de *Serial.write();* existen diversas librerías que facilitan la labor de realizar tareas gráficas de una forma rápida sin tener que conocer qué comandos concretos son necesarios. En este sentido, podemos destacar la librería "displayshield4d", descargable de http://code.google.com/p/displayshield4d, la librería "uoled-library", descargable de http://code.google.com/p/uoled-library o la librería "serial_LCD", descargable de http://embeddedcomputing.weebly.com.

Si, en cambio, su chip gráfico está en modo GFX, el chip gráfico se convierte en un dispositivo autónomo y el desarrollo de las aplicaciones gráficas se ha de realizar en un lenguaje propio de 4Dsystems, el 4DGL. Este lenguaje permite tomar el control de todas las capacidades del chip gráfico (manejo de la pantalla, del puerto

serie, de la tarjeta microSD), de manera que las aplicaciones escritas con 4DGL se ejecutan directamente en el shield, liberando a la placa Arduino de prácticamente toda la faena. Por otro lado, el lenguaje 4DGL permite exprimir las funcionalidades del chip gráfico mucho más. En la página web de 4DSystems hay abundante documentación para aprender a utilizar este lenguaje, y se pueden descargar las aplicaciones de desarrollo pertinentes. También están disponibles las herramientas para cambiar el firmware del chip y hacer que funcione en un modo u otro.

Módulos OLED de Adafruit

Podemos utilizar módulos OLED (que no son shields propiamente dichos) con Arduino. En concreto, Adafruit distribuye dos módulos monocromos, ambos de 128x32 píxeles. Los dos módulos incluyen el mismo chip (el SSD1306 de Solomon Systech), el cual puede comunicarse con la placa Arduino de dos maneras. Si lo hace vía SPI, estaremos hablando del módulo producto nº 661, y si lo hace vía I^2C estaremos hablando del módulo producto nº 931.

Ambos módulos incorporan internamente un elevador de tensión para convertir los 5 V de los pines de Arduino en hasta 12 V y poder alimentar así los píxeles de la pantalla, pero, no obstante, su circuitería electrónica solo puede ser alimentada con 3,3 V para funcionar correctamente. Afortunadamente, ambos módulos (SPI y I^2C) incorporan internamente un regulador de tensión general que rebaja los 5 V a 3,3 V, por lo que podremos conectarlos a nuestra placa Arduino sin problemas.

En el caso del módulo SPI, el cableado es muy sencillo: el pin “GND” ha de ir a tierra, el pin ”Vin” ha de ir al pin de 5 V de la placa Arduino, “DATA” debe ir al pin digital nº 9, “CLK al pin digital nº 10, “D/C” al pin digital nº 11, “CS” al pin digital nº 12 y “RST” al pin digital nº 13. En el caso del módulo I^2C, el pin “GND” ha de ir a tierra, el “Vin” al pin de 5V, el pin “SDA” al pin de entrada analógica nº 4 (en el modelo UNO) , el pin “SCL” al pin de entrada analógica nº 5 y el pin “RST” al pin digital nº 4.

Una vez hechas las conexiones, para poder utilizar ambos módulos lo más sencillo es descargar e instalar la librería que la gente de Adafruit ha desarrollado especialmente para él, disponible en https://github.com/adafruit/Adafruit_SSD1306. Esta librería incluye códigos de ejemplo para ambos modelos muy interesantes para estudiar, los cuales están accesibles dentro del menú “File->Sketchbook->Libraries->Adafruit_SSD1306” del IDE Arduino. Estos ejemplos muestran diversos textos, imágenes, rectángulos, círculos y líneas por la pantalla en diferentes posiciones y tamaños, pero atención, no funcionarán si no se ha descargado e instalado antes otra librería, la “Adafruit GFX” (ya comentada en párrafos anteriores).

Un tercer módulo OLED monocromo de Adafruit es el producto nº 938. Este módulo tiene una pantalla de 128x64 e incorpora el chip SSD1306 (como los dos anteriores módulos), por lo que se programa con las mismas librerías. Se puede comunicar con Arduino utilizando tanto el protocolo SPI como el I^2C, y también integra un regulador de voltaje interno general, por lo que también se puede conectar directamente a los pines de la placa Arduino sin problemas.

Un problema que tienen los tres módulos monocromos anteriores es que carecen de zócalo microSD o de cualquier otro tipo de sistema de almacenamiento para alojar ningún fichero de imagen. Esto obliga a tener que usar algún "truco" para poder visualizar fotografías en ellos. Concretamente, para mostrar una imagen por la pantalla primero deberemos convertirla con nuestro editor de imágenes preferido al formato BMP monocromo y redimensionarla a un tamaño igual o menor a la resolución de la pantalla OLED. El siguiente paso es transformar los bits de esa imagen a códigos numéricos (usando por ejemplo el programa –solo para Windows– "LCD Assistant", descargable desde http://en.radzio.dxp.pl/bitmap_converter o bien la aplicación online disponible en http://manytools.org/hacker-tools/image-to-byte-array, entre otras). Una vez obtenidos estos códigos, finalmente deberemos insertarlos dentro de nuestro sketch, el cual, cuando se ejecute, se encargará de generar a partir de esos códigos la imagen en tiempo real.

Afortunadamente, Adafruit ofrece un módulo OLED con pantalla RGB de 96x64 píxeles (nº de producto 684), capaz de mostrar fotografías de más de 65000 colores (en formato BMP a 24 colores) y con un zócalo microSD para almacenar esas imágenes. No obstante, este módulo funciona a 3,3 V y no incorpora ningún regulador de tensión interno, por lo que para ser conectado a la placa Arduino se necesita utilizar un componente extra: un convertidor de nivel.

Breve nota sobre los convertidores de nivel:

Un chip convertidor de nivel (en inglés, "level shifter") tiene la función de regular la tensión entre dos componentes electrónicos cuyo voltaje de trabajo es diferente. Un caso típico es la conexión entre la placa Arduino (cuyos pines de entrada/salida funcionan todos con una señal de 5 V) y una placa externa (como puede ser el módulo OLED de Adafruit comentado anteriormente, cuya alimentación y pines de entrada/salida funcionan todos a 3,3 V). Si se conectan los pines de la placa Arduino directamente a los pines del módulo OLED, estos recibirán más tensión de la soportada y se quemarán. Y en dirección contraria también hay problemas: si la placa Arduino recibe directamente señales provenientes del módulo, en ocasiones un nivel HIGH algo por debajo de los

3,3 V podría confundirse con un nivel LOW en el rango de 5 V. Por tanto, un convertidor de nivel permite "empalmar" varios pines funcionando a 5V a varios pines funcionando a 3,3 V (y viceversa) sin que aparezcan estos problemas.

Existen determinados componentes cuya tensión de alimentación es de 3,3 V pero no necesitan convertidores de nivel para funcionar, debido a que sus pines de entrada/salida son compatibles con 5 V. En esos casos, lo único que debemos procurar es alimentar la placa a la tensión adecuada y nada más. Pero hay que asegurarse primero de que, efectivamente, los pines de entrada/salida admitan tensiones de 5 V.

Existen infinidad de convertidores de nivel; si buscamos uno en formato DIP, para nuestros proyectos con Arduino nos puede venir bien el chip 75LVC245 o el 74HC4050 de Texas Instruments o el HEX4050BP de NXP, todos disponibles en Mouser, Jameco o distribuidores similares. Básicamente, disponen de un conjunto de varias patillas donde podemos conectar los pines GPIO del componente que trabaja a una tensión mayor, otro conjunto de patillas donde podemos conectar los pines GPIO del componente que trabaja a una tensión menor, un par de patillas donde recibe respectivamente las dos alimentaciones y patillas para tierra. Hay que tener en cuenta, no obstante, que un chip convertidor de nivel puede ser bidireccional o no; es decir, que puede transmitir las señales en ambos sentidos o en uno solo –generalmente de tensión mayor a menor– . Dependiendo de nuestras necesidades, elegiremos uno u otro.

También podríamos utilizar convertidores de nivel en forma de plaquitas breakout. Por ejemplo, Adafruit distribuye su producto nº 395, que dispone de hasta 8 líneas bidireccionales por banda, y el producto nº 757, con 4 líneas bidireccionales. Sparkfun por su lado, distribuye el producto nº 8745, con 4 líneas bidireccionales. Freetronics distribuye su "Logic level converter module" , que dispone de 4 líneas bidireccionales. Todos ellos funcionan perfectamente para comunicar componentes trabajando a 5 V con otros trabajando a 3,3 V (y otros voltajes, dependiendo del modelo).

En el caso particular de querer conectar canales I^2C, de manera que, por ejemplo, podamos empalmar un dispositivo I^2C funcionando a 3,3 V a los pines SDA y SCL de la placa Arduino (que funcionan a 5 V), deberíamos utilizar unos convertidores de nivel específicos para I^2C, ya que este protocolo necesita un sistema interno de resistencias "pull-up" que en otro tipo de comunicaciones como la serie (líneas RX,TX) o la SPI (líneas CS, MOSI, MISO y SCK) no es necesario. Ejemplos de estos convertidores específicos son el producto nº 757 de Adafruit y el producto

nº 8745 de Sparkfun ya nombrados, y además específicamente el producto nº 10403 de Sparkfun.

En cualquier caso, los convertidores de nivel tan solo funcionan para señales digitales, no convierten señales analógicas. Para lograr esto, se necesita un componente diferente: un amplificador operacional (comúnmente llamados “op-amp”).

Shields y módulos que incorporan pantallas TFT

Las pantallas TFT (a diferencia de las OLED, que se basan en una tecnología completamente diferente) son una versión mejorada de las pantallas LCD. Si comparamos las pantallas TFT con las OLED, veremos que las segundas son más delgadas, ligeras y flexibles, tienen un mejor ángulo de visión, muestran los colores más vívidos, tienen un mejor contraste y consumen menos. Pero en cambio, su tiempo de vida útil es mucho menor.

Adafruit distribuye su “1.8" TFT Shield” con nº de producto 802. Este shield incorpora un zócalo para insertar una tarjeta microSD (formateada en FAT16 o FAT32) con imágenes guardadas en ella y una pantalla TFT de 128x160 píxeles y 1,8 pulgadas de diagonal que permite mostrar más de 250000 colores diferentes. Además, incorpora un joystick de cinco posiciones (arriba, abajo, derecha, izquierda, selección). El controlador que lleva (el ST7735 de Sitronix) trabaja a 3,3 V, pero el shield incorpora un convertidor de nivel que permite no preocuparnos de este tema. Por tanto, para usar este shield tan solo tenemos que encajarlo sobre la placa Arduino y listo. Como solo se apropia de 4 pines digitales de nuestra placa Arduino para la transmisión de datos vía SPI (los nº 8, 10, 11 y 13), del pin analógico nº 3 para el control del joystick y de los pines digitales nº 4 y nº 12 para la gestión de la tarjeta microSD, deja aún mucho margen para poder seguir utilizando bastantes entradas y salidas de Arduino.

Para poder trabajar con este shield, necesitamos instalar dos librerías: la propia de este dispositivo llamada “Adafruit ST7735”, disponible en https://github.com/adafruit/Adafruit-ST7735-Library, y la librería “Adafruit GFX” (la verdadera responsable del dibujado de los píxeles, líneas, rectángulos, triángulos, círculos y texto). Todas estas librerías vienen con ejemplos de código muy bien documentados. Respecto a la visualización de fotografías guardadas en una tarjeta microSD, estas han de ser obligatoriamente de formato BMP de 24 colores y de un tamaño máximo igual a la resolución de la pantalla.

Si se desea, se puede utilizar un módulo independiente (una placa breakout) con la misma pantalla, chip controlador, zócalo de tarjeta microSD y circuitería reguladora de voltaje que el shield anterior, pero sin el joystick. Se programa además con la misma librería "Adafruit ST7735". Es el producto nº 358 de Adafruit. Este módulo se comunica vía SPI con la placa; solo hay que tener la precaución de conectar bien los cables de la siguiente manera:

Placa breakout	Placa Arduino
LITE	5V (para luz de fondo)
MISO	Pin digital nº 12 (si se usa SD)
SCK	Pin digital nº 13
MOSI	Pin digital nº 11
TFT_CS	Pin digital nº 10
CARD_CS	Pin digital nº 4 (si se usa SD)
D/C	Pin digital nº 9
RESET	Pin digital nº 8
VCC	5V
GND	GND

Otro módulo independiente con pantalla TFT integrada distribuido por Adafruit es el "2,2'' TFT Display", con nº de producto 797. Esta plaquita breakout también incorpora un zócalo para insertar una tarjeta microSD (formateada en FAT16 o FAT32) con imágenes guardadas en ella y una pantalla TFT de 220x176 píxeles y 2,2 pulgadas de diagonal que permite mostrar más de 250000 colores diferentes. El controlador que lleva incorporado (el HX8340B de Himax) trabaja a 3,3 V, pero el módulo incorpora un convertidor de nivel interno que permite no preocuparnos de este tema. Así pues, solo hay que tener la precaución de conectar bien los cables, así:

Placa breakout	Placa Arduino
GND	GND
VIN	5V
RST	Pin digital nº 9
SDCS	Pin digital nº 4 (si se usa SD)
CS	Pin digital nº 10
MOSI	Pin digital nº 11
SCK	Pin digital nº 13
MISO	Pin digital nº 12 (si se usa SD)

Para poder trabajar con este módulo necesitamos instalar dos librerías: la propia de este dispositivo llamada "Adafruit HX8340B", disponible en

https://github.com/adafruit/Adafruit-HX8340B, y la librería "Adafruit GFX". Todas estas librerías vienen con ejemplos de código muy bien documentados. Respecto a la visualización de fotografías guardadas en una tarjeta microSD, estas han de ser obligatoriamente de formato BMP de 24 colores y de un tamaño máximo igual a la resolución de la pantalla.

Volviendo a los shields, GorillaBuilderz distribuye su "LCD Shield", el cual es un shield que, a pesar de su nombre, no incorpora ninguna pantalla de fábrica sino que está pensada para albergar diversas pantallas TFT, tales como la "µLCD-24T(SGC)", la "µLCD-28T(SGC)" o la "µLCD-32T(SGC)", todas ellas fabricadas por 4DSystems. Estas pantallas son configurables mediante comandos hexadecimales transmitidos por el canal serie. La buena noticia es que podemos utilizar una librería específica de este shield para dibujar figuras geométricas, animaciones, texto, etc., sin tener que escribir los comandos directamente usando *Serial.write()* . Esta librería está disponible en https://github.com/gorillabuilderz/gb-arduino-libs.

Shields y módulos que incorporan pantallas TFT táctiles

Adafruit distribuye su "2,8'' TFT Touch Shield" con nº de producto 376. Este shield incorpora un zócalo para insertar una tarjeta microSD (formateada en FAT16 o FAT32) con imágenes guardadas en ella y una pantalla TFT de 240x320 píxeles y 2,8 pulgadas de diagonal que permite mostrar más de 250000 colores diferentes. Además, adherido a toda la pantalla hay un sensor resistivo que permite detectar la presión ejercida por los dedos. El controlador que lleva incorporado trabaja a 3,3 V, pero el shield incorpora un regulador de voltaje interno que permite no preocuparnos de este tema. Por tanto, para usar este shield tan solo tenemos que encajarlo sobre la placa Arduino y listo. De lo que sí que debemos preocuparnos es de que nos quedamos sin pines de entrada/salida usando este shield, ya que colocando el shield sobre la placa los cubrimos todos.

Para poder trabajar con este shield, necesitamos instalar tres librerías: la propia de este dispositivo llamada "Adafruit TFTLCD", disponible en https://github.com/adafruit/TFTLCD-Library, la librería "Adafruit GFX" (verdadera responsable del dibujado de los píxeles, líneas, rectángulos, triángulos, círculos y texto) y también la librería "Adafruit Touch Screen", disponible en https://github.com/adafruit/Touch-Screen-Library para poder detectar la "x", la "y" y la "z" (presión) de los contactos realizados con los dedos. Todas estas librerías vienen con ejemplos de código documentados, tanto para mostrar las capacidades gráficas de la pantalla como sus capacidades de respuesta táctil. Respecto a la visualización de fotografías guardadas en una tarjeta microSD, estas han de ser obligatoriamente de

formato BMP de 24 colores y de un tamaño máximo igual a la resolución de la pantalla.

De todas maneras, si es necesario utilizar pines-hembra de la placa Arduino, ya hemos visto que con el shield anterior no será posible. Una posible solución es utilizar un módulo independiente con la misma pantalla, sensor táctil y chip controlador conectado a través de cables a la placa Arduino y ubicado de tal manera (sobre una breadboard, por ejemplo), que los pines-hembra no usados de Arduino puedan estar disponibles sin nada que los tape. Este módulo existe y es el producto nº 335 de Adafruit. Funciona exactamente igual que el shield acabado de explicar excepto en que no dispone de zócalo microSD.

Este módulo se puede conectar a nuestra placa Arduino de dos maneras diferentes, las cuales están serigrafiadas cada una en un lateral. En uno de ellos aparece una tira de 20 agujeros (a soldar a una ristra de pines estándar, ya que no vienen soldados de fábrica) y en el otro aparecen dos tiras de diez agujeros, pero ambas son equivalentes. Se han de conectar básicamente tres “componentes” del módulo: la luz de fondo (“backlite”, que requiere un pin digital), la LCD propiamente dicha (que requiere 8 pines digitales para los datos y 4 pines analógicos para el envío y llegada de señales de control) y el sensor táctil (que requiere 2 pines digitales y 2 analógicos, pero se pueden reutilizar porque no interfieren con el funcionamiento de la LCD), además de la conexión de reseteo, alimentación y tierra común. Es decir:

Placa breakout	Placa Arduino
D7	Pin digital nº 7
D6	Pin digital nº 6
D5	Pin digital nº 5
D4	Pin digital nº 4
D3	Pin digital nº 3
D2	Pin digital nº 2
D1	Pin digital nº 9
D0	Pin digital nº 8
Y-	Pin digital nº 9
X-	Pin analógico nº 2
Y+	Pin analógico nº 3
X+	Pin digital nº 8
Backlite	5V
RST	RESET
RD	Pin analógico nº 0
WR	Pin analógico nº 1

Placa breakout	Placa Arduino
C/D	Pin analógico nº 2
CS	Pin analógico nº 3
3-5V	5V
GND	GND

Por lo tanto, quedan libres los pines digitales nº 0 y nº 1 (serie), nº 10, 11, 12 y 13 (SPI) y los pines analógicos nº 4 y nº 5 (I^2C).

Si quisiéramos mostrar fotografías, en el shield venía incorporado un zócalo microSD que la plaquita breakout no tiene. La solución es añadir a nuestro circuito otra plaquita breakout independiente que dé soporte a un zócalo microSD (como por ejemplo el producto nº 254 de Adafruit) y la podamos comunicar con el módulo TFT a través de la placa Arduino. Concretamente, si elegimos el producto nº 254, tendremos que conectar sus pines etiquetados como “DI”, “DO”, “SCK” y “CS” a los pines-hembra de Arduino nº 11, 12, 13 y 10, respectivamente (es decir, establecer una comunicación SPI), además de sus pines “5V” y “GND”.

Otro módulo (que no shield) con pantalla táctil interesante es el “Smart GPU” de Vizictechnologies. Su pantalla es de 320x240, 2,4 pulgadas y puede mostrar más de 250000 colores. Incorpora además un zócalo microSD formateable en FAT16 y FAT32. Tanto la pantalla como el sensor táctil como la tarjeta microSD se gestionan mediante una librería propia, descargable de su web, junto con varios ejemplos de código. Internamente se comunica vía serie con el receptor/transmisor TTL-UART del ATmega328P, por lo que en realidad todas las instrucciones de esa librería no son más que simples comandos hexadecimales enviados vía serie, pero las funciones (de tipo *milcd.drawline()* o *milcd.drawCircle()*) que incorpora su librería propia convierten la programación de este shield en algo muy sencillo e intuitivo. Los únicos pines que se han de utilizar para conectar este shield con una placa Arduino son el de alimentación (¡ha de ser de 3,3V!), tierra, RESET, TX (al RX del Arduino) y RX (al TX del Arduino).

Otro módulo con pantalla táctil es el ezLCD (producto nº 11000 de Sparkfun), el cual tiene la particularidad de poderse conectar directamente vía USB a un computador (tal como si fuera un simple lápiz de memoria) para poder almacenar en su tarjeta microSD las imágenes y los vídeos deseados. Su programación se realiza mediante el envío de comandos específicos (detallados ampliamente en su completa guía de usuario) a través del mismo cable USB, utilizando un software de tipo “terminal serie”.

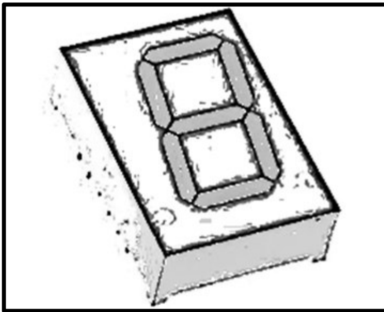
IteadStudio, por su parte, también distribuye varios módulos TFT táctiles con zócalo SD incorporado, de diferentes tamaños y resoluciones de pantalla. Son los llamados módulos “ITDB02”, entre los cuales podemos encontrar por ejemplo el módulo “ITDB02-2.4E” (con una pantalla de 2,4 pulgadas y 320x240 píxeles de resolución a 65000 colores) el “ITDB02-2.8” (similar, pero con una pantalla de 2,8 pulgadas) o el “ITDB02-2.2” (con pantalla de 2,2 pulgadas), entre muchos más. IteadStudio también distribuye un shield llamado “ITDB02 Arduino shield” que permite acoplar sin problemas cualquiera de los tres módulos anteriores (los cuales trabajan a 3,3 V) a una placa Arduino UNO.

Todos los módulos “ITDB02” pueden ser programados mediante la librería “UTFT”, descargable en http://www.henningkarlsen.com . Esta librería también puede ser utilizada para trabajar con módulos TFT de otro fabricante; concretamente los módulos de la serie llamada “TFT01” de Elecfreaks. No obstante, la librería “UTFT” tan solo maneja el aspecto gráfico de las pantallas: para utilizar su capacidad táctil, necesitamos descargar e instalar además la librería “ITDB02_Touch”, también disponible en la web de Henning Karlsen. Alternativamente, una librería para los módulos “ITDB02” no tan completa como las dos anteriores pero que reúne en una sola tanto funcionalidades de dibujado como de respuesta táctil es la disponible en http://code.google.com/p/itdb02. En todo caso, si deseamos además gestionar la tarjeta microSD (formateada en FAT16) ubicada en el zócalo que ofrecen estos módulos, deberemos utilizar otra librería extra específica, tal como la “tinyFAT”, disponible también en la web de Henning Karlsen.

IteadStudio distribuye además varios shields TFT táctiles completos (con pantalla y zócalo SD incorporados), como el “ITEAD 2.4 TFT LCD Touch shield”, el “ITEAD 2.8 TFT LCD Touch shield” o el “ITEAD 3.2 TFT LCD Touch shield”. Desgraciadamente, no ofrecen una librería específica para Arduino: tan solo códigos de ejemplo (con funciones reutilizables, eso sí).

Shields que incorporan displays “7-segmentos”

Un display de 7 segmentos no es otra cosa que 7 LEDs conectados entre sí de manera que encendiendo algunos de ellos y apagando otros podemos mostrar la forma de diferentes números. Tal vez su uso pueda parecer redundante al lado de una LCD, pero si en nuestro proyecto solamente deseamos mostrar números o se necesita una visibilidad realmente grande, los 7-segmentos son la mejor opción. Además, consumen mucha menos corriente que un LCD con luz de fondo.



Podríamos utilizar 7-segmentos individuales (como por ejemplo el producto nº 8546 de Sparkfun), o incluso conjuntos de 4 7-segmentos (como por ejemplo el producto nº 9480), pero ciertamente, el cableado necesario para ponerlos en marcha (y los sketches resultantes) son algo engorrosos, ya que por cada 7-segmentos debemos controlar 7 LEDs individualmente. Pero no solo eso: de hecho, si observamos la cantidad de pines que tiene un 7-segmentos, veremos que suelen tener diez, ubicados en dos filas de cinco: siete de estos pines sirven para iluminar cada segmento, pero hay otro más para iluminar el punto decimal y hay otros dos pines más (generalmente, los situados en el centro de cada fila y numerados con la posiciones 3 y 8), que se han de conectar ambos a tierra (si el 7-segmentos es de tipo "cátodo común") o bien ambos a 5 V (si el 7-segmentos es de tipo "ánodo común"). Resumiendo: es necesario utilizar más de 8 pines-hembra de nuestra placa Arduino para manejar un solo 7-segmentos. Por tanto (aunque existen otras soluciones), lo más cómodo para nosotros será utilizar shields que ya tengan incorporado un conjunto de 4 dígitos 7-segmentos y que permitan un uso mucho más eficiente (y una programación más fácil) de nuestra placa Arduino.

Por ejemplo, NootropicDesign ofrece su "Digit Shield" listo para acoplar a nuestra placa Arduino, la cual solo monopoliza cuatro pines de entrada/salida digital (los nº 2,3,4 y 5). Este shield puede ser programado por una librería descargable de su propia web muy sencilla y flexible, la cual contiene diferentes funciones (del tipo *mipantalla.setValue()* o *mipantalla.setDigit()*, entre otras) para mostrar los valores numéricos de varias maneras.

Otro shield similar es el "7-segment shield" de Gravitech, que además de 4 dígitos de 7-segmentos también incorpora "de regalo" un sensor de temperatura, un LED multicolor y una memoria EEPROM extra de 128 KBytes. Todo ello monopoliza 4 pines de la placa Arduino. Desgraciadamente, para programarla en nuestro sketch Arduino, debemos utilizar directamente la librería "Wire" oficial de Arduino, ya que no hay ninguna librería que nos oculte la complejidad de la comunicación I^2C requerida. De todas formas, nos ofrecen un código de ejemplo para que veamos cómo se ha de hacer.

También podemos encontrar útil la placa LEDuino de Cal-Eng, que no es un shield sino una placa propiamente dicha, capaz de sustituir a la placa Arduino oficial. Incorpora el chip ATmega328P funcionando con el bootloader de Arduino y es

programable a través de FTDI mediante el entorno de desarrollo oficial de Arduino, pero tiene la particularidad de incorporar 4 dígitos 7-segmentos de forma totalmente integrada.

Si no queremos usar ningún shield o placa con los 7-segmentos integrados, podemos usar módulos con 4 dígitos de 7-segmentos de una forma relativamente cómoda. Por ejemplo, Gravitech distribuye su "I^2C 4 digit 7-segment display", con cuatro pines acoplables en una breadboard, que tal como su nombre indica, se comunica mediante I^2C con la placa Arduino y se ha de programar igualmente con la librería oficial "Wire" (se proporciona código de ejemplo).

Otro módulo interesante de 4 dígitos es el "7-segment serial display" (producto nº 9764 de Sparkfun), el cual incorpora un chip ATmega168 que puede comunicarse con el receptor/transmisor TTL-UART de la placa Arduino. Para ello, se ha de conectar el pin del módulo etiquetado como "RX" al pin TX de la placa, además de conectarlo a tierra y a alimentación (de entre 2,6 V y 5,5 V). Si se desea, podemos hacer que el módulo utilice el protocolo SPI en vez del canal serie para comunicarse: en ese caso debemos conectar sus pines etiquetados como "MOSI", "SCK" y "CSN" a los pines "MOSI", "SCK" y "SS" de la placa Arduino (además de conectarlo a tierra y la alimentación). Sea cual sea el método de comunicación, este módulo es controlado mediante comandos hexadecimales enviados vía *Serial.write()* y los números a mostrar son especificados mediante una línea *Serial.print()* por cada segmento individual uno tras otro, de izquierda a derecha.

Ejemplo 5.6: A continuación se muestra un código de ejemplo que implementa un contador de segundos y minutos usando la comunicación serie (se recomienda consultar el datasheet del producto para conocer la variedad de comandos posibles):

```
const int txPin = 1;  // Pin de salida serie
const int rxPin = 0;  // No se usa, pero se ha de poner
void setup() {
       pinMode(txPin, OUTPUT);
/*Inicializo la comunicación con el display. La velocidad en bits/s
se puede cambiar de forma permanente (hasta nuevo cambio) si se le
envía el comando 0x7F seguido del código correspondiente al valor
deseado. Para más información, consultar el datasheet. */
       Serial.begin(9600);
/*Comando que pone los cuatro 7-segmentos en blanco.
  Cada "x" representa un segmento */
      Serial.print("xxxx");
//El comando 0x74 indica que se configura el brillo
```



```
       Serial.write(0x7A);
/*Concretamente, establece el brillo bastante
bajo (los valores posibles son de 0x00 hasta 0xFE)*/
       Serial.write(0x0F);
/*El comando 0x77 indica que se configurarán las comas decimales */
       Serial.write(0x77);
/*Concretamente, las apaga todas. Si se quiere mostrar la
coma decimal, poner: 0x10, pero hay más posibilidades */
       Serial.write(0x0F);
      //Comando que reinicia el display
      Serial.write(0x76);
}
void loop() {
/*decenaminutos=segmento de más a la izquierda
minutos=segundo segmento de más a la izquierda
decenasegundos= tercer segmento de más a la izquierda
segundos = cuarto segmento de más a la izquierda */
  byte segundos, minutos, decenasegundos, decenaminutos;
  segundos++;
  if(segundos > 9) { segundos = 0; decenasegundos++; }
  if(decenasegundos > 5) { decenasegundos = 0; minutos++; }
  if(minutos > 9) { minutos = 0; deciminutos++; }
  if(decenaminutos > 9) {decenaminutos = 0;Serial.print("xxxx"); }
  if(decenaminutos > 0) {
    Serial.print(decenaminutos);  Serial.print(minutos);
    Serial.print(decenasegundos); Serial.print(segundos);
  } else if(minutos > 0) {
      Serial.print("x");            Serial.print(minutos);
      Serial.print(decenasegundos); Serial.print(segundos);
  } else if(decenasegundos > 0) {
      Serial.print("x");            Serial.print("x");
      Serial.print(decenasegundos); Serial.print(segundos);
  } else {
      Serial.print("x");            Serial.print("x");
      Serial.print("x");            Serial.print(segundos);
  }
  delay(1000);
}
```

Si usamos el módulo anterior y necesitamos transformar un número de varias cifras en dígitos individuales para poderlos manipular mejor (por ejemplo, para mostrar un valor obtenido de un sensor, o el número de pulsaciones realizadas sobre un botón, o una simple cuenta atrás que finalice en 0), una buena idea es convertir

ese número en un objeto String, para así utilizar funciones como *micadena.length()* (para saber de cuántas cifras es ese número) y sobre todo *micadena.charAt()* (para seleccionar un dígito individual concreto de entre los cuatro posibles, y entonces imprimirlo con *Serial.print()*). Se deja como ejercicio.

Otro módulo de 4 dígitos 7-segmentos es el "TWI 7-seg Display" de Akafugu. Tal como su nombre indica, ha de ser controlado mediante el protocolo I^2C, por lo que solo necesita cuatro cables para conectarse a una placa Arduino: alimentación (puede ser tanto 3,3 V como 5 V), tierra, línea SDA y línea SCL. Para trabajar con él se ha de utilizar la librería descargable de https://github.com/akafugu/twidisplay . Esta misma librería es la que también se necesita para poder utilizar otro producto de Akafugu, el "TWIDisplay 8-digit LCD", el cual es una pantalla de 8 dígitos 14-segmentos (controlable también vía I^2C mediante los mismos 4 cables: alimentación -3,3 V o 5 V-, tierra, SDA y SCL). Los 14-segmentos son capaces de formar todas las figuras del alfabeto, tanto en minúsculas como en mayúsculas.

Otro módulo de 4 dígitos 7-segmentos es el producto nº 878 de Adafruit (y otros similares con diferentes colores). También se comunica con la placa Arduino mediante I^2C, por lo que tiene 4 pines: el etiquetado como "CLK" se ha de conectar al pin SCL de la placa (el pin de entrada analógico nº 5) y el etiquetado como "DAT" al pin SDA de la placa (el pin de entrada analógico nº 4). Además, está el pin de tierra y el de alimentación (a 5 V). Para trabajar con él cómodamente nos podemos descargar de https://github.com/adafruit/Adafruit-LED-Backpack-Library la librería "Adafruit LED Backpack" e instalarla de la forma habitual (aunque recordemos que esta librería necesita tener instalada también la librería "Adafruit GFX").

Matrices de LEDs

Lo bueno de la librería "Adafruit LED Backpack" es que la podemos utilizar sin ningún cambio en otro tipo de displays ofrecidos por Adafruit formados por múltiples LEDs: los módulos de matrices de LEDs 8x8 tales como el producto nº 870 (y otros similares de diferentes colores, incluyendo la matriz bicolor nº 902). Esto es debido a que todos estos productos incorporan el mismo chip controlador HT16K33. Igualmente, requieren cuatro cables para conectarse a una placa Arduino: alimentación (5 V), tierra, línea SDA y línea SCL. Para aprender a programarlos se puede consultar el código de ejemplo llamado "matrix88" (o bien el llamado "bimatrix88", para la matriz bicolor) que viene junto con la librería "Adafruit LED Backpack". En estos códigos además se muestra el uso de las funciones de la librería "Adafruit GFX" que permiten dibujar figuras, líneas, etc.

Otros módulos similares al anterior son los productos nº 759 y nº 760 de Sparkfun. Ambos incorporan también una matriz de 8x8 LEDs pero, a diferencia del modelo de Adafruit, los de Sparkfun se comunican vía SPI. En el producto nº 759 la matriz de bicolor (rojo-verde) y en el producto nº 760 la matriz es de tipo RGB, (permitiendo mostrar hasta siete colores diferentes). No requieren ninguna librería en particular para ser controlados desde una placa Arduino: tan solo el uso de la librería SPI oficial de Arduino. En la página del producto nº 760 se ofrece un código de ejemplo.

Los módulos anteriores, ya sean los de Sparkfun o el de Adafruit, disponen de la característica de poderse conectar unos a otros en “cascada”, de forma que se puedan formar conjuntos de matrices de LEDs de grandes dimensiones.

No obstante, si se quieren manejar matrices de LEDs grandes, una solución realmente sencilla es utilizar matrices multicolores de elevadas dimensiones ya de fábrica. Un ejemplo es la “Dot Matrix Display” de 32x16 (512) LEDs de Freetronics. Su conexión a Arduino es tremendamente fácil mediante el acoplamiento de una plaquita breakout (incluida en el producto) a los pines digitales nº 6, 7, 8, 9, 10, 11, 12 y 13 de la placa Arduino. A esa plaquita breakout se ha de enchufar un cable de cinta de 22 cm (incluido en el producto) que va a parar en el otro extremo al conector pertinente de la pantalla (es el situado más a la izquierda: el conector de la derecha está pensado para conectarlo a otra pantalla de este tipo y hacer así un panel en cadena). Una vez alimentada cada pantalla (mediante su correspondiente conector presente en su dorso) con una fuente externa de 5 V y 2,8A, ya está.

Para poder controlar esta pantalla de Freetronics desde Arduino, hemos de utilizar una librería propia, la “DMD”: https://github.com/freetronics/DMD. Esta librería permite la escritura de caracteres y cadenas en diferentes fuentes, así como el pintado de píxeles individuales y el dibujado de líneas y figuras geométricas. En la web del producto se pueden descargar claros códigos de ejemplo, así como la guía de instalación detallada.

Otras pantallas “gigantes” son el producto nº 420 (16x32 píxeles, 512 LEDs RGB) y el nº 607 (32x32 píxeles, 1024 LEDs RGB) de Adafruit, muy parecidas a la pantalla de Freetronics. Desgraciadamente, no vienen acompañadas de ninguna plaquita breakout que facilite la conexión del cable de cinta a los pines-hembra de la placa Arduino, así que el cableado es algo engorroso (aunque está bien explicado en la página web del producto). Ambas pantallas se programan mediante la librería “Adafruit RGB Matrix Panel”, descargable de aquí: https://github.com/adafruit/RGB-matrix-Panel (se requiere también la instalación de la librería “Adafruit GFX”).

Adafruit también ofrece el producto nº 555 (16x24 LEDs rojos), que no es más que un acoplamiento de 6 matrices 8x8. Utiliza también cables de cinta, por lo que la conexión a la placa Arduino no es del todo directa. Es programable mediante la librería "HT1632" (llamada así por el chip controlador que incorporan), descargable de aquí: https://github.com/adafruit/HT1632 , además de la "Adafruit GFX".

Otras pantallas gigantes que utilizan el mismo chip controlador HT1632 son las "24x16 LED Dot Matrix Unit Boards" de Sure Electronics, con código de producto DE-DP11111 si los 384 LEDs son de 3 mm y de color verde, con código DE-DP11112 si son de 3 mm y de color rojo, con código DE-DP11211 si son de 5 mm y de color verde o con código DE-DP11212 si son de 5 mm y de color rojo. Para programar estas pantallas de SureElectronics, podemos utilizar la librería "HT1632" disponible en https://github.com/milesburton/HT1632.

Por otro lado, también podemos utilizar shields específicos que ya incorporan matrices de LEDs. De esta forma, simplemente acoplando un shield de este tipo a nuestra placa Arduino podemos empezar a trabajar. Dentro de esta categoría podemos encontrar el "LoL Shield" que distribuye Adafruit (productos nº 274, 286, 493, 494… según el color de los LEDs), el cual dispone de una matriz 9x14 programable mediante la librería descargable de http://code.google.com/p/lolshield.

Otro shield es el "LED Matrix Shield" de Hobbytronics, que ofrece una matriz 8x8 de color rojo; para programarla no se requiere ninguna librería en especial (en la página del producto se pueden consultar diversos códigos de ejemplo).

Otro shield es el "Colors Shield" de IteadStudio, que ofrece un zócalo para poder colocar una matriz 8x8 RGB, la cual no necesita de ninguna librería especial para ser programada (en la página del producto se pueden consultar diversos códigos de ejemplo), aunque existe una librería no oficial que facilita su uso y que puede ser descargada de aquí: https://github.com/lincomatic/Colorduino.

Incluso, si no queremos usar el combo Arduino+shield, podemos utilizar placas propiamente dichas (en sustitución de la mismísima placa Arduino) especialmente diseñadas para alojar una matriz 8x8 RGB. Estas placas son compatibles con el entorno de programación oficial de Arduino y se pueden programar como si fueran una placa UNO estándar, ya que incorporan el microcontrolador ATmega328P con el bootloader de Arduino y un conector FTDI. Un ejemplo es la placa Colorduino de IteadStudio, funcionalmente idéntica a su shield "Colors Shield" pero en forma de placa. Otro ejemplo es la placa Rainbowduino de

Seeedstudio, que utiliza la librería "Rainbowduino Library" (descargable de la web del producto) para la gestión de los LEDs.

USO DE LA MEMORIA EEPROM

La librería oficial EEPROM consta tan solo de dos funciones:

EEPROM.write(): escribe un byte en una celda especificada de la EEPROM. Tiene dos parámetros: el primero es el número de celda donde se escribirá el byte (existen 1024, empezando por la número 0); el segundo es el valor a escribir, el cual ha de ser un valor entero entre 0 y 255. No tiene valor de retorno.

Hay que tener en cuenta que una escritura de la EEPROM tarda 3,3 ms en completarse, y sobre todo, que esta memoria tiene una vida media de 100000 ciclos de escritura/borrado, así que se tiene que ir con cuidado con esto.

EEPROM.read(): lee un byte de la celda especificada. Tiene un parámetro: el número de celda de donde se lee el byte (existen 1024, empezando por la número 0). Su valor de retorno es el valor del byte leído. Las celdas que no han sido escritas previamente guardan un valor de 255.

Ejemplo 5.7: Un código autoexplicativo de esta librería es el siguiente:

```
#include <EEPROM.h>
int i,celda = 0;
byte valor;
void setup()
{
	Serial.begin(9600);
/*Escribo la mitad de la memoria EEPROM (los primeros 512 bytes) con
valores que coinciden con su índice. Se puede observar cómo cuando se
quiere escribir un valor mayor de 255, se empieza de 0 otra vez.*/
	for (i = 0; i < 512; i++){
		EEPROM.write(i, i);
	}
}
void loop()
{
```

```
	//Leo el byte ubicado en una celda
	valor = EEPROM.read(celda);
	//Muestro el nº de la celda y el valor acabado de leer
	Serial.print(celda);  Serial.print("\t");
	Serial.print(valor);  Serial.println();
	//Me muevo a la celda siguiente
	celda = celda + 1;
/* Si ya he llegado a la última celda que contiene valores escritos
en el setup(), vuelvo otra vez al principio. */
	if (celda == 512){
		celda = 0;
	}
	delay(50);
}
```

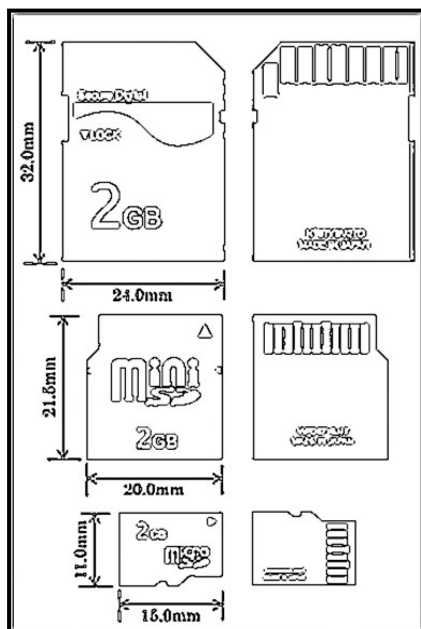
Si se necesitara disponer de más memoria EEPROM de la que ofrece la placa Arduino UNO por defecto, se le puede conectar un módulo externo, como el "EEPROM Data Storage Module" de DFRobot (basado en el chip AT24C256 de Atmel), el cual aporta 32 Kbytes extra y se puede conectar a través del estándar I^2C. En la página oficial de DFRobot se ofrece un completo tutorial de su uso.

USO DE TARJETAS SD

Características de las tarjetas SD

Secure Digital (SD) es un formato de tarjeta de memoria no volátil diseñada para ser usada en dispositivos portátiles (teléfonos móviles, cámaras digitales, computadores portátiles, etc.) cuya especificación es mantenida por la SD Card Association (https://www.sdcard.org), entidad que engloba muchos fabricantes de hardware.

El estándar SD incluye 4 "familias", las cuales son las siguientes (por orden cronológico de aparición): las tarjetas originales (SDSC –Standard capacity–); las tarjetas de alta capacidad (SDHC –High capacity–, que permiten almacenar hasta 32 Gbytes); las tarjetas de capacidad extendida (SDXC –Extended capacity–, que permiten almacenar hasta 2048 GBytes); y las tarjetas que combinan almacenamiento de datos con funciones de entrada/salida (SDIO).



Los dispositivos que alojan las tarjetas SD siempre son compatibles "hacia atrás" (es decir: si el dispositivo es compatible con SCXC, también lo será con SCHC y SDSC, por ejemplo). No obstante, cada una de estas familias redefine sus conexiones internas, por lo que una tarjeta SCXC (por ejemplo) no puede ser utilizada dentro de dispositivos que no sean explícitamente compatibles con esa familia en concreto.

Desgraciadamente, las tarjetas SDXC se comercializan preformateadas con el sistema de ficheros privativo y patentado exFAT de Microsoft, por lo que algunos dispositivos es posible que no sean capaces de leer su contenido. En estos casos, para que estas tarjetas sean reconocidas, se deben formatear con otro sistema de ficheros, generalmente FAT32. De hecho, la mayoría de tarjetas SD de las otras familias se suelen adquirir en cualquier tienda local de electrónica ya formateadas en el sistema FAT32. El FAT32 es el sistema de ficheros más extendido en tarjetas SD debido a que la gran mayoría de aparatos electrónicos que utilizan este tipo de tarjetas (cámaras, móviles, etc.) son capaces de reconocer este formato sin problemas.

Si necesitamos formatear una tarjeta SD en el sistema FAT32 o bien FAT16 (bien porque la adquirimos sin formato o bien porque le queremos quitar el que tiene) debemos conectarla mediante un lector de tarjetas apropiado a un computador para que este la reconozca como un dispositivo de almacenamiento más. Una vez reconocida, el proceso concreto a seguir dependerá del sistema operativo instalado en el computador y de las aplicaciones especializadas proporcionadas por éste. De todas formas, desde la SD Card Association recomiendan usar la utilidad de formateo desarrollada por ellos mismos (solo disponible para sistemas Windows y Mac OS X), descargable gratuitamente de https://www.sdcard.org/downloads .

Las formas físicas con las que se comercializan las cuatro familias son tres: "SD", "miniSD" y "microSD", tal como se puede ver en la ilustración anterior. En general (salvo alguna excepción aislada), existen combinaciones de todas las familias con todas las formas. En el mercado podemos encontrar adaptadores que permiten el uso de una tarjeta más pequeña en un zócalo grande y también dispositivos USB que contienen zócalos de múltiples tamaños para poder insertar allí tarjetas SD (y de

otros formatos) y posibilitar que en un computador las pueda leer, si es que este no lleva ya incorporados estos zócalos de fábrica.

Por otro lado, la velocidad mínima garantizada de transferencia de datos viene definida por la "clase" de la tarjeta (la cual puede ser de cualquier familia). Así, encontramos tarjetas de Clase 2 (2 MBytes/s), de Clase 4 (4 MBytes/s), de Clase 6 (6 MBytes/s) o de Clase 10 (10 MBytes/s). La velocidad máxima puede variar bastante entre distintas tarjetas: el modo Ultra-High Speed (UHS) –implementado en algunas tarjetas SDHC y en casi todas las SDXC–, permite velocidades teóricas de hasta aproximadamente 100 MBytes/s en su versión "I" y de hasta aproximadamente 300 MBytes/s y en su versión "II", siempre y cuando el dispositivo donde se aloje la tarjeta en cuestión sea capaz de gestionar estas velocidades también.

Como la mayoría de los formatos de tarjeta de memoria, el SD está cubierto por numerosas patentes y marcas registradas, y solo se puede licenciar a través de la SD Card Association. Para solucionar este problema, un método común es utilizar el modo SPI/MMC (que las tarjetas SD implementan siempre) para emular el comportamiento de las antiguas tarjetas MultiMediaCard. Concretamente, en este modo algunos de los pines de la tarjeta son usados como las líneas MOSI, MISO, SCK y SS del protocolo SPI (además de usar otros pines para conexiones a alimentación, tierra o nada). De esta forma se puede utilizar una conexión que, aunque más lenta, es compatible con los puertos SPI de muchos microcontroladores. Otros modos de transferencia soportados por las tarjetas SD son el llamado modo "un-bit" (propietario) y el modo "cuatro-bit" (que soporta transferencias paralelas de cuatro bits, también propietario), los cuales utilizan los pines de la tarjeta de otra forma.

La librería SD

Antes de conocer las instrucciones que provee la librería oficial SD, hay que tener en cuenta una serie de consideraciones que pasamos a comentar.

Esta librería solamente permite nombre de ficheros con una longitud máxima de 8 caracteres más 3 caracteres para su extensión. Además, los nombres de ficheros son case-insensitive; es decir: es lo mismo "mifichero.txt" que "mifichero.TXT" que "mifichero.tXt".

Los nombres de ficheros pasados como parámetros en las distintas funciones de esta librería pueden ser en realidad rutas completas (es decir, pueden incluir el nombre de las distintas carpetas a las que se ha de acceder para llegar al fichero en cuestión). En cualquier caso, esta ruta separa cada subcarpeta intermedia con el

signo de la barra / , como por ejemplo así: “carpeta/fichero.txt” (al igual que ocurre de hecho con los sistemas Linux y MacOSX).

Por defecto la carpeta de trabajo es la carpeta “raíz” de la tarjeta (la carpeta llamada “/”). Esto hace que especificar el nombre de un fichero “a secas” sea equivalente a especificar ese nombre precedido de una barra. Es decir, “/fichero.txt” equivale a “fichero.txt”.

La librería asume por defecto que la comunicación entre el microcontrolador de la placa Arduino UNO y la tarjeta SD (esté montada esta sobre un shield o módulo cualquiera) se establece mediante el protocolo SPI a través de los pines digitales nº 10 (pin SS) y nº 11, 12 y 13. No obstante, si la tarjeta SD escogida utilizara otro pin para la funcionalidad SS (por el diseño hardware concreto del shield/módulo donde está ubicada) es posible configurar la librería para reconocer dicho otro pin como pin SS en vez del nº 10. Por ejemplo, en el caso de la placa Arduino Ethernet o el shield Arduino Ethernet, el pin SS utilizado por la tarjeta SD por defecto es el número 4, ya que el número 10 se reserva la comunicación interna con el chip Ethernet. De todas maneras, es muy importante tener en cuenta que, aunque no se utilice el pin 10 como SS, este deberá ser configurado siempre explícitamente como salida digital (OUTPUT), o si no la librería SD no funcionará.

La primera instrucción que debemos ejecutar antes de cualquier otra de la librería SD (normalmente dentro de la función “setup()” es:

SD.begin(): inicializa la librería y la tarjeta SD. Opcionalmente, admite un parámetro para especificar el número de pin de la placa Arduino que la librería utilizará como canal SS en la comunicación SPI con la tarjeta SD. Si este parámetro no se especifica, por defecto la librería utilizará el pin número 10. Hemos de tener en cuenta, no obstante, que si usamos la placa Arduino Ethernet (o el shield con el mismo nombre), el pin que debemos utilizar como pin SS es el número 4, por lo que deberemos invocar esta función así: `SD.begin(4);`. En todo caso, hay que tener en cuenta que aunque la librería SD utilice un pin diferente del número 10, ese pin número 10 siempre ha de configurarse como salida digital (OUTPUT) para que todo funcione correctamente. El valor de retorno de esta función es de tipo booleano: valdrá 1 si la orden se ejecuta correctamente y 0 si ocurre algún error.

También tenemos estas otras funciones de la librería SD:

SD.exists(): comprueba si un fichero (o carpeta) pasado por parámetro existe en la tarjeta SD. Como único parámetro tiene el nombre o ruta del fichero (o carpeta) del cual se quiere comprobar su existencia. Su valor de retorno es de tipo booleano: valdrá 1 si existe, 0 si no.

SD.mkdir(): crea una carpeta en la tarjeta SD. Como único parámetro tiene el nombre o ruta de la carpeta que se quiere crear. Si la carpeta ya existiera, esta función no hace nada. Si se especifican carpetas intermedias que no existen, también se crean (por ejemplo, si escribimos `SD.mkdir("a/b/c");` se crearán la carpeta "a", y dentro de esta "b", y dentro de esta, "c". Su valor de retorno es de tipo booleano: valdrá 1 si la orden se ejecuta correctamente, 0 si ocurre algún error.

SD.rmdir(): elimina una carpeta de la tarjeta SD. Esa carpeta ha de estar vacía para poder ser borrada. Como único parámetro tiene el nombre o ruta de la carpeta que se quiere eliminar. Su valor de retorno es de tipo booleano: valdrá 1 si el borrado se ha realizado correctamente o 0 si ha ocurrido algún error, pero si la carpeta especificada no existiera, el valor de retorno está no determinado.

SD.remove(): elimina un fichero de la tarjeta SD. Como único parámetro tiene el nombre o ruta del fichero que se quiere eliminar. Su valor de retorno es booleano: valdrá 1 si el borrado se ha realizado correctamente o 0 si ha ocurrido algún error; si el fichero especificado no existiera, el valor de retorno está no determinado.

Y sobre todo:

SD.open(): "abre" un fichero en la tarjeta SD. Abrir un fichero significa poder empezar a leer o bien editar su contenido. En el caso concreto de que se abra para escritura un fichero que no exista, será creado. Es posible tener múltiples ficheros abiertos a la vez.

La importancia de esta función radica en que devuelve un objeto de tipo File, que representa el fichero abierto. La variable que recoja este objeto devuelto (la llamaremos "mifichero") debe declararse previamente en nuestro sketch como de tipo "File", así : `File mifichero;`. La importancia del objeto File devuelto por *SD.open()* está en que este objeto contiene a su vez un conjunto de instrucciones que permiten manipular individualmente ese fichero que representa.

Como primer parámetro *SD.open()* tiene el nombre o ruta del fichero que se quiere abrir. Como segundo parámetro (opcional) tiene el modo en el que se quiere abrir el fichero; puede tener dos valores posibles: o bien la constante FILE_READ (el modo por defecto), que indica que el fichero se abre en solo lectura empezando por su principio, o bien la constante FILE_WRITE, que indica que el fichero se abre para leer y también escribir en él empezando por su final (para así añadir contenido nuevo sin sobrescribir el existente).

Una vez creado el objeto de tipo File mediante la función *SD.open()*, a partir de aquí podemos manipular este fichero en particular con diferentes instrucciones propias de ese objeto. Suponiendo que ese objeto de tipo File lo llamamos "mifichero", las funciones que nos permiten escribir datos en una tarjeta son:

mifichero.print(): escribe el dato pasado como parámetro (que puede ser tanto de tipo "char" o cadena como también entero) dentro del fichero "mifichero", el cual lógicamente debe haber sido abierto en modo escritura. Si el dato es numérico entero, es tratado como caracteres ASCII (es decir, el número 123 es escrito como tres caracteres, '1' ,'2' y '3'). Opcionalmente, como segundo parámetro se puede especificar (si el valor a enviar es entero) una de las constantes predefinidas BIN, HEX o DEC, para elegir el sistema de numeración empleado para representar ese dato (binario, hexadecimal o decimal –por defecto–, respectivamente). Su valor de retorno es de tipo "byte" e indica el número de bytes escritos; su uso es opcional.

mifichero.println(): hace exactamente lo mismo que *mifichero.print(),* pero además, siempre añade automáticamente al final de los datos escritos dos caracteres: el de retorno de carro (código ASCII nº 13) y el de nueva línea (código ASCII nº 10).

mifichero.write(): escribe el dato que se haya pasado como parámetro dentro del fichero "mifichero" (que debe haber sido abierto en modo escritura). Esta función puede escribirse de dos maneras: si el dato a grabar solo ocupa un byte o si ocupa más. En el primer caso, esta función aceptará un único parámetro que deberá ser de tipo "byte" o "char". En el segundo caso, esta función puede seguir aceptando un único parámetro, que será una cadena de caracteres, o bien aceptar dos parámetros: el nombre de un array de datos de tipo "byte" o "char" y el número de elementos que se escribirán en él (el cual no tiene por qué coincidir con el número total de elementos del array). Su valor de retorno es de tipo byte e indica el número de bytes escritos; su uso es opcional.

Sea como sea, hay que tener en cuenta, no obstante, que las funciones *mifichero.print(), mifichero.println()* y *mifichero.write()* no graban físicamente el dato en la tarjeta: para ello es necesario ejecutar una de las dos funciones siguientes. Es decir, hay que asegurarse siempre de utilizar alguna de estas dos funciones después de utilizar las funciones de escritura de datos para que se guarden realmente en la tarjeta.

mifichero.close(): cierra el fichero "mifichero". Antes se asegura de escribir físicamente en la tarjeta cualquier dato pendiente de grabar en ese fichero. No tiene ni parámetros ni valor de retorno.

mifichero.flush(): se asegura de escribir físicamente en la tarjeta cualquier dato pendiente de grabar en el fichero "mifichero". Esto también se realiza automáticamente cada vez que el fichero es cerrado (con *mifichero.close()*). No tiene ni parámetros ni valor de retorno.

Para leer datos de la tarjeta SD, disponemos de las siguientes funciones:

mifichero.available(): comprueba si hay algún byte todavía por leer dentro de "mifichero". Su valor de retorno (de tipo "int") es el número de bytes aún disponibles para leer. No tiene parámetros.

mifichero.read(): lee un byte del fichero "mifichero" y avanza al siguiente byte, de manera que la próxima ejecución de *mifichero.read()* lea ese siguiente byte. Su valor de retorno es el valor del byte (o carácter) leído, o -1 si ya no hay más bytes por leer. No tiene parámetros.

mifichero.peek(): lee un byte del fichero "mifichero" sin avanzar al siguiente. Es decir, ejecuciones seguidas de *mifichero.peek()* devolverán siempre el mismo valor, hasta que se ejecute *mifichero.read()* –momento en el que se avanzará hacia el siguiente byte–. Su valor de retorno es el valor del byte (o carácter) leído, o -1 si ya no hay más bytes por leer. No tiene parámetros.

También disponemos de funciones para posicionarse en cualquier byte dentro del interior de un fichero:

mifichero.size(): devuelve el tamaño del fichero "mifichero" en bytes (es un valor de tipo "unsigned long"). No tiene parámetros.

mifichero.position(): **d**evuelve la posición actual del cursor dentro del fichero "mifichero". Es decir, el lugar dentro de ese fichero a partir del cual se leerá o escribirá el siguiente byte. Esta posición indica en realidad el número de un byte, contando byte a byte desde el principio del fichero. La primera posición es la número 0. Este valor devuelto es de tipo "unsigned long". No tiene parámetros.

mifichero.seek(): cambia la posición del cursor dentro del fichero "mifichero" a la ubicación dada como parámetro. Esta posición indica en realidad el número de un byte, contando byte a byte desde el principio del fichero. Debe valer entre 0 y el tamaño en bytes del fichero (ambos incluidos), y es de tipo "unsigned long". Su valor de retorno es de tipo booleano: vale 1 si la orden se ejecuta correctamente o 0 si ocurre algún error.

Finalmente, también disponemos de funciones especializadas en trabajar dentro de una carpeta determinada con múltiples ficheros (uno tras otro):

mifichero.isDirectory(): las carpetas (también llamadas a menudo "directorios") son tipos especiales de ficheros. Esta función informa de si "mifichero" es un fichero "estándar" o bien es una carpeta mediante su valor de retorno booleano: si vale 1 "mifichero" es una carpeta y vale 0, no. No tiene parámetros.

undirectorio.openNextFile(): devuelve un objeto File que representa el siguiente fichero encontrado (por orden de creación) existente dentro del directorio "undirectorio". Si no hay ya ninguno más, devolverá "false". No tiene parámetros.

mifichero.name(): devuelve el nombre (en formato de 8 caracteres más los 3 de la extensión) del objeto "mifichero". No tiene ni parámetros ni valor de retorno.

undirectorio.rewindDirectory(): retrocede hasta el primer fichero de la lista de ficheros existentes dentro de la carpeta "undirectorio". Se suele usar conjuntamente con la instrucción *undirectorio.openNextFile()*. No tiene ni parámetros ni valor de retorno

Una vez explicadas todas las funciones pertenecientes a la librería SD, veamos unos cuantos ejemplos.

Ejemplo 5.8: El siguiente código muestra cómo escribir una cadena en una tarjeta SD:

```
#include <SD.h>
File miarchivo;
void setup(){
/* Establecemos el pin n° 10 como salida digital, o si no la librería
SD no funcionará. Esto es lo que hace la siguiente línea, aunque no
la hayamos estudiado todavía.*/
	pinMode(10, OUTPUT);
	//Si ha habido un error, se aborta el programa
	if (SD.begin(10) == 0) {
		return;
	}
	//Se abre el fichero para poder escribir en él
	miarchivo = SD.open("test.txt", FILE_WRITE);
	//Si el fichero se abre ok, se escribe una frase y se cierra
	if (miarchivo != 0) {
		miarchivo.println("Probando 1, 2, 3.");
		miarchivo.close();
	}
}
void loop(){}
```

Si en vez de escribir en la tarjeta una frase literal, hubiéramos escrito los datos obtenidos por uno (o más) sensores, ya tendríamos un sketch que nos permitiría guardar de forma permanente esta información y así poder, por ejemplo, realizar un estudio posterior pormenorizado utilizando una hoja de cálculo en un computador.

Ejemplo 5.9: He aquí cómo hacer para leer el dato grabado en el anterior ejemplo.

```
#include <SD.h>
File miarchivo;
void setup(){
	Serial.begin(9600);
	pinMode(10, OUTPUT);
	if (SD.begin(10) == 0) {return;}
	miarchivo = SD.open("test.txt");
	if (miarchivo != 0) {
//Lee del fichero hasta que ya no quede ningún byte por leer
		while (miarchivo.available() > 0) {
//El byte leído lo muestro en el "Serial monitor"
			Serial.write(miarchivo.read());
		}
```

```
//Es bueno cerrar el fichero para ahorrar RAM una vez listo
             miarchivo.close();
       }
}
void loop() {}
```

Ejemplo 5.10: El siguiente ejemplo muestra cómo escribir en un fichero sobrescribiendo su contenido anterior. El truco utilizado ha sido borrarlo y volverlo a crear, aunque una manera algo más elegante sería emplear *miarchivo.seek()* para situar el cursor en el byte 0:

```
#include <SD.h>
File miarchivo;
void setup(){
       pinMode(10, OUTPUT);
       if (SD.begin(10) == 0) {
             return;
       }
       if (SD.exists("test.txt") !=0){
             SD.remove("test.txt");
       }
       miarchivo = SD.open("test.txt", FILE_WRITE);
       if (miarchivo != 0) {
//miarchivo.seek(0);  (Una manera alternativa de sobrescribir)
             miarchivo.println("Probando 1, 2, 3.");
             miarchivo.close();
       }
}
void loop(){}
```

Ejemplo 5.11: El siguiente código muestra cómo detectar en cada arranque de Arduino si, al querer crear un nuevo fichero, ya hay grabado de antes otro fichero con el mismo nombre. Si ese es el caso, el sketch cambia dinámicamente el nombre del nuevo fichero a crear por otro, para no modificar en absoluto el fichero previamente grabado. Haciendo esto, en cada arranque de Arduino generaremos un fichero diferente, independiente de los generados en anteriores arranques.

```
#include <SD.h>
File miarchivo;
void setup(){
       pinMode(10, OUTPUT);
       SD.begin(10);
```



```
      //Empiezo con este nombre
      char nombrefichero[]="LOGGER00.CSV";
/*El bucle sirve para ir probando combinaciones de nombres de
ficheros hasta que se encuentre una que no está utilizada todavía */
      for (byte i = 0; i < 100; i++) {
/*El truco para probar combinaciones es cambiar los dos últimos
caracteres del nombre del fichero para que sean consecutivamente
“00”,“01”,”02”,”03”…”10”,”11”,”12”…hasta “99” (por tanto, solo
podremos tener hasta 100 ficheros diferentes). Para conseguir estas
combinaciones, utilizamos el resultado y el resto de una división de
un contador entre diez. A ello se le suma el carácter ASCII ‘0’
(equivalente al número 48) para convertir el dígito obtenido en ambas
operaciones en su carácter ASCII correspondiente (es decir, para
convertir por ejemplo el número 6 en el carácter “6”, que tiene
código ASCII número 54) */
         nombrefichero[6] = i/10 + '0';
         nombrefichero[7] = i%10 + '0';
         //Si no existe la combinación actual como nombre de fichero…
         if (!SD.exists(nombrefichero)) {
            //…creo el nuevo fichero con esa combinación como nombre
            miarchivo = SD.open(nombrefichero, FILE_WRITE);
            break;  //…y no sigo probando de crear ningún fichero más
         }
      }
      /*En cada reinicio de Arduino, tendré un
      nuevo fichero con la misma frase */
      if (miarchivo != 0) {
            miarchivo.println("Probando 1, 2, 3.");
             miarchivo.close();
       }
}
void loop(){}
```

Ejemplo 5.12: El siguiente código (algo más complejo) muestra la lista de los nombres y tamaños de los ficheros presentes dentro de la carpeta indicada (concretamente la carpeta “raíz”). Se puede ver el uso de una función propia recursiva, la cual muestra los ficheros encontrados tabulados según la subcarpeta a la que pertenezcan.

```
#include <SD.h>
File raiz;
File entrada;
int i;
void setup() {
```

```
       Serial.begin(9600);
       pinMode(10, OUTPUT);
       if (SD.begin(10) ==0) {return; }
       raiz = SD.open("/");
       printDirectory(root, 0);
}
void loop(){}

void printDirectory(File dir, int numTabs) {
   while(true) { //Bucle infinito
       entrada =  raiz.openNextFile();
       /*No hay más ficheros, así que acabo la ejecución. Si la
       función se invocó recursivamente, vuelvo a la anterior*/
       if (entrada == 0) { break; }
       //Pongo los tabuladores oportunos antes del nombre del fichero
       for (i=0; i<numTabs; i++) {
              Serial.print('\t');
       }
       //Escribo el nombre del fichero
       Serial.print(entrada.name());
/*Si resulta que es una subcarpeta, aumento en uno la tabulación y
vuelvo a llamar a la misma función para que haga todo el recorrido de
ficheros por esa subcarpeta */
       if (entrada.isDirectory() !=0 ) {
              Serial.println("/");
              printDirectory(entrada, numTabs+1);
       //Si no, se muestra al lado del nombre el tamaño del fichero
       } else {
              Serial.print("\t\t");
              Serial.println(entrada.size(), DEC);
       }
   }
}
```

Dentro del conjunto de sketches de ejemplo que vienen "de fábrica" con el IDE Arduino, hay uno en especial que es muy interesante, el "CardInfo". Su código es relativamente complejo y utiliza objetos internos avanzados de la librería que no hemos estudiado, pero su utilidad radica en ejecutarlo para comprobar las características de la tarjeta SD que tengamos conectada en ese momento a la placa Arduino. Concretamente, obtiene el tipo de tarjeta (si es SD o SDHC), el tamaño en KBytes y MBytes de la partición FAT16 o FAT32 detectada, y la lista de ficheros almacenados (mostrando su nombre, fecha de última modificación y tamaño en bytes).

Shields que incorporan zócalos microSD

Para poder utilizar tarjetas SD, podemos utilizar shields que incorporen el zócalo y la circuitería pertinente. Todos ellos han de incorporar internamente un regulador de voltaje a 3,3 V (o usar uno externo) para no dañar la tarjeta. Ejemplos de ello son:

El “microSD shield” de Sparkfun. Se puede programar mediante librería oficial de Arduino, aunque utiliza el pin nº 8 como SS.

El “SD card shield” de Seeedstudio. Dispone de zócalo un zócalo SD además del microSD, seleccionable mediante un conmutador. Se programa mediante la librería “SdFat”, descargable desde http://code.google.com/p/beta-lib.

El “Stackable SD card shield” de Iteadstudio. Se puede programar mediante la librería oficial de Arduino, o bien mediante la librería “TinyFAT”, descargable desde http://henningkarlsen.com/electronics.

El “microSD 2GB module” de Cooking-Hacks. Se adquiere con tarjeta incluida. Tiene la particularidad de poderse conectar bien a los pines SPI de la placa Arduino UNO o bien a los pines ICSP. Se puede programar mediante la librería oficial de Arduino.

El “Memoire shield” de Logos Electromechanical LLC. Se programa mediante la librería “SdFat” mencionada anteriormente. Incluye además una zona de prototipado y un reloj de tiempo real (un RTC, concretamente el módulo DS1307 de Maxim). Es programable mediante cualquier librería compatible, como por ejemplo http://code.google.com/p/ds1307new.

El “Data logging shield” de Adafruit. La tarjeta microSD se programa mediante la librería oficial de Arduino y utiliza los pines SPI estándar (10, 11, 12 y 13). Incluye además una zona de prototipado y un módulo RTC (concretamente el DS1307 de Maxim), comunicado con la placa Arduino a través de los pines I^2C , programable mediante la librería RTCLib de la propia Adafruit (https://github.com/adafruit/RTClib) y alimentado por una pila botón CR1220 incluida. El shield funciona a 3,3V (para no dañar la tarjeta) gracias a que incluye un regulador propio independiente del de la placa Arduino; con él se consigue más intensidad de corriente (necesaria para las escrituras de ficheros) y una mayor estabilidad y exactitud en la tensión. No obstante, se vende sin ensamblar, por lo que es necesario soldar los componentes.

Módulos que incorporan zócalos microSD

También se pueden incluir en nuestros proyectos módulos independientes que no sean shields. Un ejemplo puede ser el "microSD breakout board" de Adafruit , el cual tiene ofrece los pines 5V (a conectar al pin 5V de la placa Arduino), GND (a conectar a algún pin GND) y los pines necesarios para la comunicación SPI: CLK (a conectar al pin 13), DO (a conectar al pin 12), DI (a conectar al pin 11) y CS (a conectar al pin 10). Gracias a un regulador de voltaje interno rebaja la tensión de trabajo de la tarjeta a los 3,3 V que esta necesita, y se programa con la librería SPI oficial de Arduino.

Otro módulo muy parecido al anterior (que también se conecta mediante el protocolo SPI y por tanto, también se programa mediante la librería oficial de Arduino) es el producto nº 544 de Sparkfun. Otro similar es también el módulo "ADPmicroSD" de Propox.

Por otro lado, un módulo digno de destacar es el OpenLog de Sparkfun (nº de producto 9530), el cual es una placa breakout con un zócalo SD que se comunica a través del canal serie (pines RX y TX) con la placa Arduino. Su tarea es registrar todos los datos recibidos por ese canal serie, pero lo original es que mediante ese mismo canal (es decir, usando *Serial.write()*) se pueden enviar una serie de comandos propios que permiten crear ficheros, borrarlos, listarlos, etc., directamente.

USO DE PUERTOS SERIE SOFTWARE

Lo primero que debemos hacer para poder utilizar una pareja de pines extra como RX y TX mediante la librería oficial SoftwareSerial es declarar una variable global de tipo "SoftwareSerial", que representará dentro de nuestro sketch a este nuevo canal serie "virtual". La declaración se ha de realizar usando la siguiente sintaxis (suponiendo que llamamos "miserie" a dicha variable): *SoftwareSerial miserie(npintx,npinrx);* donde "npintx" en realidad es un valor numérico que indica el pin de la placa Arduino que hará la función TX (es decir, el que enviará datos al exterior) y "npinrx" en realidad es un valor numérico que indica el pin de la placa que hará la función RX (es decir, el que recibirá datos del exterior). Un ejemplo de declaración podría ser, pues: `SoftwareSerial miserie(2,3);`.

Una vez creado el canal *miserie* con la línea anterior, lo primero que debemos hacer (normalmente dentro de la función "setup()") es abrir la conexión mediante la instrucción **miserie.begin()**, la cual funciona exactamente igual que la ya conocida *Serial.begin()*.

A partir de aquí ya podemos utilizar un conjunto de funciones que nos permitirán gestionar ese canal *miserie* en particular. Por ejemplo, podemos hacer uso de **miserie.available(), miserie.peek()** y **miserie.read()** para recibir datos, y **miserie.print()**, **miserie.println()** y **miserie.write()** para enviarlos. Todas ellas son funciones equivalentes a las ya vistas cuando se trató el objeto Serial. Las únicas funciones novedosas que aporta un objeto SoftwareSerial respecto al objeto Serial son las siguientes:

miserie.listen(): en el caso de que exista más de un objeto SoftwareSerial, activa el objeto llamado "miserie" para que pueda recibir datos (o dicho de otra forma, "pone a la escucha" al objeto "miserie"). Solo un objeto SoftwareSerial puede recibir datos en un determinado momento: los datos que lleguen al resto de puertos que no escuchen son desechados. No tiene ni parámetros ni valor de retorno.

miserie.isListening(): comprueba si "miserie" es el puerto serie virtual (de entre los posibles existentes) que está ahora mismo escuchando. Su valor de retorno es booleano: valdrá "true" si está escuchando y "false" si no. No tiene parámetros.

miserie.overflow(): comprueba si la cantidad de datos recibidos ha sobrepasado el límite de almacenamiento del buffer de entrada de "miserie". El tamaño de los buffers de los objetos SoftwareSerial es de 64 bytes, al igual que el del objeto Serial hardware. Su valor de retorno es booleano: valdrá "true" si se ha superado el límite y "false" si no. No tiene parámetro.

Ejemplo 5.13: El sketch siguiente es un ejemplo sencillo que ilustra el uso de esta librería, donde se va alternando el uso de dos canales serie por software:

```
#include <SoftwareSerial.h>
//TX = pin digital 10, RX = pin digital 11
SoftwareSerial canalUno(10, 11);
//TX = pin digital 8, RX = pin digital 9
SoftwareSerial canalDos(8, 9);
void setup(){
    Serial.begin(9600);
    canalUno.begin(9600);
    canalDos.begin(9600);
}
void loop(){
    char dato;
```

```
    //Primero se escucha por el puerto Uno
    canalUno.listen();
    Serial.println("Datos recibidos por el puerto Uno:");
/*Mientras se reciban datos por puerto Uno, se leen byte a byte y
se van enviando al puerto serie hardware para verlos por el "Serial
monitor" */
    while (canalUno.available() > 0) {
        dato = canalUno.read();
        Serial.write(dato);
    }
//Línea en blanco para separar los datos recibidos de cada puerto
    Serial.println();
    //Ahora se escucha por el puerto Dos
    canalDos.listen();
    Serial.println("Datos recibidos por el puerto Dos:");
    while (canalDos.available() > 0) {
        dato = canalDos.read();
        Serial.write(dato);
    }
    Serial.println();
}
```

Ejemplo 5.14: El siguiente ejemplo lo podríamos realizar mediante los canales serie hardware, pero lo veremos utilizando la librería SoftwareSerial para demostrar que los canales "emulados" funcionan exactamente igual. Para probarlo deberemos tener dos placas Arduino UNO. La idea es comunicarlas mutuamente y transferir datos por el canal serie que las interconecta, de tal forma que la información generada por una placa (que podría haber sido obtenida por ejemplo de algún sensor, aunque en el código de ejemplo está escrita explícitamente para simplificar la demostración) se transmita a la otra placa (que podría servir para activar por ejemplo algún actuador, aunque en el código de ejemplo simplemente se muestra por el "Serial monitor" y ya está). A continuación, mostramos los códigos de ambas placas. Tal como se puede ver, en ambas placas hemos convertido los pines nº 2 y nº 3 en sus pines TX y RX, y viceversa.

```
Primer Arduino:
#include <SoftwareSerial.h>
SoftwareSerial emisor(2,3);
int i = 0;
void setup() {
  emisor.begin(9600);
}
```

```
void loop(){
  emisor.print(i);
  emisor.print (", ");
  i++;
  delay(500);
}

Segundo Arduino:
#include <SoftwareSerial.h>
SoftwareSerial receptor(3,2);
void setup() {
  receptor.begin(9600);
  Serial.begin(9600);
}
void loop() {
  if (receptor.available() > 0) {
    Serial.print(receptor.read());
  }
}
```

Es muy importante que para que el ejemplo anterior funcione, las tierras de ambas placas estén compartidas y así tener una referencia común.

Si quisiéramos tener un control más completo y flexible de los datos transferidos entre dos Arduinos ya sea utilizando los pines TX y RX hardware o ya sea utilizando la librería SofwareSerial (o incluso utilizando el protocolo I^2C), podríamos utilizar una librería muy práctica, la "Arduino Easy-Transfer", descargable de aquí: https://github.com/madsci1016/Arduino-EasyTransfer. Recomiendo consultar los sketches de ejemplo que vienen incluidos junto con ella para conocer su uso.

USO DE MOTORES

Conceptos básicos sobre motores

La mayoría de los motores funcionan gracias al principio de inducción. Esto quiere decir que cuando un cable conduce corriente, se genera automáticamente un campo magnético alrededor de él. Si colocamos una bobina (es decir, un cable enrollado) por la que pasa corriente entre dos polos imantados norte y sur, esta bobina será atraída por un polo y repelida por el otro (o viceversa), debido al campo magnético que ella misma genera. Cuanta más intensidad de corriente atraviese la bobina, más grande será el campo magnético generado y por tanto más grande será

la atracción o repulsión. Si colocamos la bobina además sobre un eje que puede girar, la rotación de la bobina (y por tanto, del campo magnético generado por ella) hará que vaya siendo atraída y repulsada alternativamente por cada polo exterior, generando así un "rebote" de la bobina entre los polos que dará lugar a un movimiento circular ininterrumpido del eje (mientras circule corriente por la bobina).

Existen varios datos a tener en cuenta cuando se utilizan motores en nuestros circuitos: uno de ellos es el voltaje al que pueden funcionar eficientemente, señalado por el fabricante. Con ese voltaje el motor podrá girar a su máxima velocidad, a más voltaje corre el riesgo de quemarse. A menor voltaje, el motor girará a menor velocidad.

Otro dato importante es el consumo de corriente que tienen. Este depende sobre todo de la carga que están arrastrando: a más carga, más corriente necesitan. Cada motor tiene una "corriente de paro", que es la corriente que consume cuando su giro se para por una fuerza opuesta. La corriente de paro es mucho más grande que la corriente de giro, que es la corriente consumida cuando no hay carga. Puede haber motores que consuman durante un breve tiempo casi su corriente de paro al arrancar, debido a la inercia de pasar de estar parados a moverse. La fuente de alimentación debería poder ofrecer la corriente de paro y algo más para evitar problemas. Si no es el caso, entonces se debe utilizar transistores (o más radicalmente, relés) para amplificar la corriente aportada por la fuente y así alimentar el motor convenientemente.

Otro dato es la resistencia eléctrica que ofrece el motor (como cualquier otro componente eléctrico), medida en ohmios. Otro dato más es la velocidad de giro del motor, normalmente medido en "rpm" (revoluciones por minuto).

Otro dato es el torque (también llamado par) del motor. El torque es una medida de la fuerza de empuje (de "tracción") que tiene el motor. Concretamente, representa la fuerza que tiene que realizar una rueda colocada sobre el eje del motor para poder arrastrar una carga situada a una cierta distancia de ese eje y que ejerza una fuerza opuesta al giro (normalmente, su peso). Si, por ejemplo, tenemos una carga que ejerce un peso de 1 N situada a 1 m del eje de giro, el torque necesario para moverla es de 1 N·m. Si esa misma carga la tenemos a 0,5 m, el torque será de 0,5 N·m (por tanto, el motor ha de realizar "menos esfuerzo" para moverla). Fijarse que ese mismo torque se obtendría también de situar a 1 m una carga con la mitad del peso (0,5 N). El torque que viene en los datasheets es el "torque de paro", que es el torque máximo que puede realizar el motor antes de no poder seguir girando el eje debido a la oposición de la carga: cuanto más torque tenga el motor, más cargas

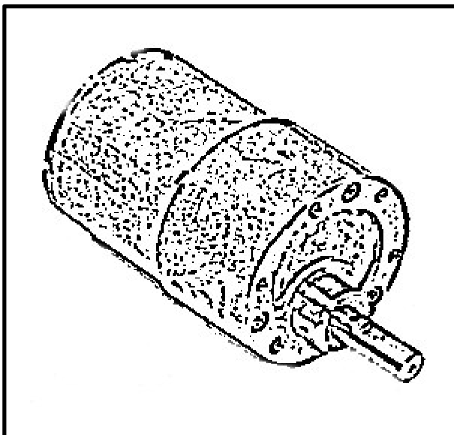
pesadas podrá rotar. Desgraciadamente, los fabricantes de motores no han estandarizado las unidades de medida del torque de paro, con lo que a veces nos podemos encontrar con datasheets que muestran esta magnitud en Kg·cm (kilogramo por centímetro), en vez de usar Newtons como sería lo físicamente correcto.

Por otro lado, comentar que todos los circuitos inductivos (como los motores), además de inducir (es decir, "crear") un campo magnético gracias a una corriente eléctrica, también funcionan en el sentido contrario. Es decir, si existe un campo magnético variable (ha de ser así) donde hay una bobina, se induce automáticamente una corriente eléctrica a través de ella. Por lo tanto, si tenemos un motor girando y lo apagamos, el campo magnético que aún existe durante un breve tiempo inducido por la bobina rotatoria (y que es variable porque la bobina aún no ha parado de girar) hace que se genere una corriente en sentido contrario al de la corriente utilizada para mover el motor. Esta corriente puede ser muy intensa y dañar la electrónica, por lo que casi siempre veremos un diodo conectado en paralelo al motor para (al estar polarizado en inversa) parar esta corriente. Este diodo se suele llamar diodo "fly-back".

Tipos de motores

Cuando se quieren mover cosas con un microcontrolador, hay básicamente tres tipos de motores útiles: motores DC, servomotores y motores paso a paso (steppers).

Los motores DC



Los motores DC (del inglés "Direct Current", corriente continua) son los más simples. Tienen dos terminales; cuando uno se conecta a una fuente de alimentación continua y el otro se conecta a tierra el motor gira en una dirección. Si se intercambia la conexión de los terminales (el que estaba conectado a tierra pasa a estar conectado a la fuente, y viceversa), el motor girará en la dirección contraria. Cuanta más intensidad de corriente atraviese el motor (es decir, cuanto más voltaje se le aplique, si suponemos su resistencia constante), girará a más velocidad de una forma casi linealmente proporcional.

En general, los motores DC realizan un consumo eléctrico bastante elevado para conseguir la velocidad de giro adecuada. Esto quiere decir que muchas veces el pin de "5 V" de nuestra placa Arduino no será suficiente, y el motor deberá ser alimentado a partir de una fuente externa, o bien mediante un amplificador de corriente (como un transistor).

Usualmente, los motores DC son capaces de girar hasta varios millares de rpms. No obstante, no tienen un torque demasiado elevado. Si se desea aumentar este, se puede conectar al motor un conjunto de engranajes (lo que se llama un "reductor" o "caja reductora"); el precio a pagar es la reducción de la velocidad máxima de giro. Los motores que incorporan este sistema son llamados motores "gearhead" o "gearmotors".

Sparkfun distribuye varios motores "gearhead" con diferentes características de dimensiones, peso, velocidad, torque y consumo. Por ejemplo, tenemos los productos nº 8910, 8911, 8912 y 8913. Si queremos motores DC sin reductor, en Sparkfun distribuyen el producto nº 9608 y en Adafruit el nº 711. Otro tipo de motores DC son los llamados "motores lineales", los cuales son capaces de generar movimiento de tracción sobre un riel, pero estos no los veremos.

Un gran inconveniente de los motores DC es que, aunque podemos controlar fácilmente la velocidad del motor, el sentido del movimiento siempre es el mismo. Ya hemos dicho que cambiar el sentido del movimiento del motor DC (ya sea rotativo o lineal) implica alternar la ubicación de los terminales de la fuente de alimentación, cosa que es bastante poco práctica en la realidad. Para conseguir esto de una forma sencilla, lo que deberemos hacer es conectar el motor a un conjunto de transistores (o relés) –en todo caso, cuatro como mínimo– dispuestos de una determinada forma comúnmente llamada "puente H". Este nombre proviene de la figura con la que las conexiones de este circuito de transistores aparecen dibujadas en los esquemas eléctricos.

Existen muchas implementaciones concretas de "puentes H" ampliamente tratadas en los libros de electrónica, pero está fuera del alcance de este libro discutir su diseño. Lo que sí haremos es utilizar "puentes H" ya integrados dentro de chips. Estos chips ahorran el trabajo de montar el diseño "a mano" y simplemente ofrecen una serie de patillas donde conectar los terminales del motor convenientemente según lo que marque su datasheet concreto.

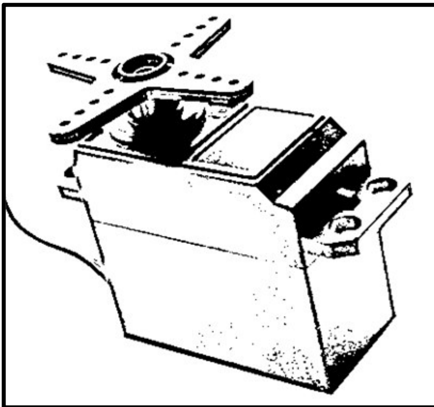
Un ejemplo muy utilizado de estos "puentes H" integrados es el chip L298 de STMicroelectronics, que es precisamente el chip que incorpora en Arduino Motor

Shield oficial. En realidad, este chip incluye dos “puentes H”, por lo que con él se puede controlar la velocidad y sentido de giro de dos motores DC (sin sobrepasar los 2A de corriente por cada puente, que es la máxima admitida) o incluso conectar ambos “puentes H” en paralelo para obtener así un solo “puente H” con el doble de capacidad de corriente.

El chip L298 se puede adquirir en la tienda oficial de Arduino o en la mayoría de distribuidores listados en el apéndice A (por ejemplo, en Sparkfun es el producto nº 9479). En Sparkfun también se puede adquirir con nº 9670 una placa breakout que incluye este chip, para una mayor comodidad de uso. Otro chip similar es el L293, el cual es más sencillo de conectar a una breadboard pero admite menos corriente de salida (0,6A) y disipa peor el calor; además su disposición de patillas no es compatible con la del L298. Una alternativa al chip L293 con su misma disposición de pines pero permitiendo el doble de corriente de salida (1,2 A) es el chip SN754410.

El estudio práctico de los motores DC lo relegaremos al siguiente capítulo, donde estudiaremos el funcionamiento de las entradas y salidas digitales en una placa Arduino.

Los servomotores



Los servomotores –también llamados “servos”– son motores “gearhead” (por tanto, motores DC con engranajes que limitan la velocidad pero aumentan el torque) que incorporan además un potenciómetro y cierta circuitería de control para poder establecer la posición del eje del motor de forma precisa. Es decir, su eje no gira libremente (como lo hace el de los motores DC) sino que rota un determinado ángulo, indicado a través de una señal de control. Lo que hace especial a un servo es, por tanto, que podemos ordenarle que gire una cantidad de grados concreta, cantidad que dependerá de la señal de control enviada en un momento dado por (por ejemplo) un microcontrolador programado por nosotros. Los servos son muy comunes en juguetes y otros dispositivos mecánicos pequeños (como por ejemplo el control de la dirección de un coche teledirigido), pero también sirven gestionar el movimiento de timones, pequeños ascensores, palancas, etc.

Los servomotores disponen normalmente de tres cables (a diferencia de los motores DC y "gearhead", que tienen solo dos): uno para recibir la alimentación eléctrica (normalmente de color rojo), otro para conectarse a tierra (normalmente de color negro o marrón, según el fabricante) y otro (el cable de control, normalmente de color blanco, amarillo o naranja) que sirve para transmitir al servo, de parte del microcontrolador, los pulsos eléctricos –de una frecuencia fija de 50 Hz en la gran mayoría de servomotores– que ordenarán el giro concreto de su eje. El cable de alimentación ha de conectarse a una fuente que pueda proporcionar 5 V y al menos 1 A. El cable de tierra ha de conectarse lógicamente a la tierra común del circuito. El cable de control debe conectarse a algún pin digital de la placa Arduino, por el cual se enviarán los pulsos que controlarán el desplazamiento angular del eje. A diferencia de los demás motores DC, para cambiar el sentido de giro del eje de los servos no es necesario invertir la polaridad de su alimentación, por lo que no es necesario incluir ningún "puente H".

Los conectores del servo son ligeramente diferentes según el fabricante, pero todos suelen ser compatibles para el uso en breadboards y similares. No obstante, hay que tener en cuenta que los distribuidos en la tienda oficial de Arduino son conectores de tipo Tinkerkit, ya que están específicamente pensados para ser conectados al "Sensor Shield" de Tinkerkit (un shield especialmente equipado con diferentes conectores Tinkerkit para enchufar allí diferentes sensores y actuadores compatibles sin necesidad de breadboard).

Los servomotores que utilizaremos en los proyectos de este libro son los más pequeños (los de tipo llamado "hobby"). Su voltaje de trabajo es de entre 5 V y 7 V, así que con la propia placa Arduino los podemos alimentar. No obstante, el consumo eléctrico de un servo es proporcional a la carga mecánica que soporta su eje (es decir, un servo consume más cuanta más "fuerza" –técnicamente, torque– necesite generar para contrarrestar la masa de los objetos colocados sobre su eje y que se oponen a su giro). Esto significa que, en la práctica, dependiendo de la carga que se le coloque al servo, será necesario utilizar una fuente externa de 5 V adicional independiente para proporcionarle una alimentación separada de la ofrecida a través de la placa Arduino (pero con la tierra común siempre). La fuente adicional también será necesaria cuando se empleen más de dos servos en nuestros circuitos, tengan la carga que tengan.

La magnitud del desplazamiento angular del eje de un servomotor está determinada por la duración de los pulsos de la señal de control. Concretamente, si el valor ALTO (5 V) del pulso se mantiene durante 1,5 milisegundos, el eje del servo se ubicará en la posición central de su recorrido. Como los servos estándar permiten

mover su eje en ángulos dentro de un rango entre 0 y 180 grados, esta posición central suele corresponder a 90 grados respecto al origen. Es decir: si al servomotor se le envía una señal con pulso de 1,5 ms, el eje girará hasta estar situado en un ángulo de 90 grados respecto al origen (por tanto, a mitad de su recorrido total). Mientras la señal de control recibida por el servomotor sea siempre la misma, este mantendrá la posición angular de su eje: si la duración del pulso de la señal varía, entonces el servomotor girará hasta la nueva posición.

Si el ancho del pulso está entre 1,5 y 2 milisegundos, el eje del servo se moverá hasta una posición angular proporcional entre 90 y 180 grados del origen. Por ejemplo: si lo quisiéramos situar a 150 grados, la longitud del pulso debería ser de 1,83 ms; si quisiéramos situarlo a 180 grados (ángulo máximo), la longitud del pulso debería ser de 2 ms (duración máxima).

Si el ancho del pulso está entre 1 y 1,5 milisegundos, el eje del servo se moverá a una posición angular proporcional entre 0 y 90 grados del origen. Por ejemplo: si lo quisiéramos situar a 30 grados, la longitud del pulso debería ser de 1,17 ms; si quisiéramos situarlo a 0 grados (es decir, el propio origen), la longitud del pulso debería ser de 1 ms (duración mínima).

De todas formas, existen modelos de servos que llegan hasta los 210 grados de rotación, por lo que, en todo caso, se recomienda consultar los datos ofrecidos por el fabricante para conocer las características de cada servo en particular.

Por otro lado, hemos de destacar también la existencia de los servomotores de rotación continua, los cuales son algo especiales. En realidad, se comportan más como motores “gearhead” que como servos propiamente dichos, porque no se les puede establecer su ángulo de giro, pero, en cambio, sí su velocidad de giro. Lo más interesante es que podemos cambiar su sentido del giro sin necesidad de ninguna circuitería extra (como serían los “puentes H” o similares).

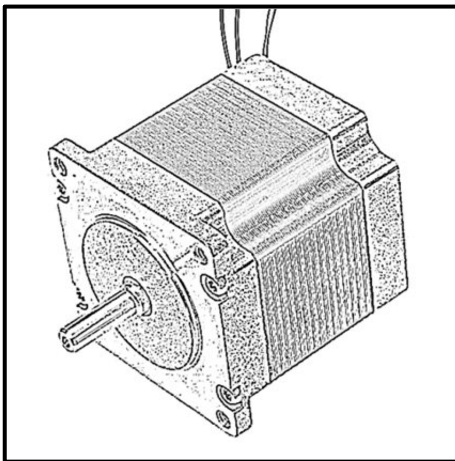
La señal de control de un servomotor de rotación continua está formada por pulsos cuadrados, generalmente a una frecuencia de 50 Hz (por tanto, igual que la de los servomotores estándar). Cuando la duración del valor ALTO (5 V) de un pulso se mantiene durante 1,5 ms, un servomotor de rotación continua permanece parado. A medida que aumenta esa duración, su velocidad de giro aumenta en un determinado sentido, hasta llegar a la velocidad máxima cuando el valor ALTO del pulso llegue a durar un máximo de 1,7 ms (generalmente). Si, por el contrario, la duración del valor ALTO del pulso es menor de 1,5 ms, el giro se producirá en el sentido contrario, y su velocidad irá en aumento a medida que la longitud del pulso disminuya, hasta llegar a

durar un mínimo de 1,3 ms (generalmente). Por ejemplo, si un servomotor de este tipo recibe una señal con un pulso de 1,525 ms, girará lentamente en un sentido, y si recibe una señal con un pulso de 1,575 ms girará en el mismo sentido pero algo más deprisa; si recibe un pulso de 1,395 ms, girará más deprisa aún, pero en el otro sentido.

Los servos se pueden adquirir fácilmente en cualquier distribuidor electrónico como los que están listados en el apéndice A. También se puede consultar http://www.servocity.com, un sitio web especializado en servomotores y motores en general. Como ejemplo de productos podemos nombrar los ofrecidos por Adafruit: el nº 169 (microservo), el nº 155 (servo estándar), el nº 154 (servo de rotación continua) o el kit nº 171, ideal para empezar a trabajar con motores, ya que incluye el microservo y el servo estándar anteriores, además de un motor DC, un motor stepper y el Arduino Motor Shield. Sparkfun ofrece por su parte los productos nº 9065 (microservo), nº 9064 (servo estándar) y los nº 9347 y nº 10189 (servos de rotación continua).

Destaquemos por otro lado la iniciativa de http://www.openservo.com, proyecto que implementa ofrece a la comunidad la posibilidad de construir un servomotor digital totalmente libre. Para ello pone a disposición tanto los diseños de PCB como el firmware necesario para poder fabricar un servomotor propio.

Los motores paso a paso



Los motores “paso a paso” (en inglés, “stepper motors”) se diferencian del resto de motores vistos anteriormente en que no giran continuamente, sino que lo hacen un número de “pasos” muy concretos. Un paso es el movimiento mínimo que puede hacer de una vez el motor, y su magnitud es configurable: puede consistir en una rotación tan grande como 90 grados, o bien un movimiento angular tan pequeño como 2 grados, según lo que interese en cada momento. Por ejemplo, para completar un giro de 360 grados, se necesitarían 4 pasos en el primer caso y 180 en el segundo. Como este diseño permite un control muy preciso de los movimientos del eje, estos motores son usados ampliamente en todo

dispositivo donde la posición exacta del motor sea un requisito necesario, como por ejemplo impresoras, discos duros, etc.

Los motores paso a paso se mueven usualmente mucho más lentamente que los motores DC, ya que hay un límite máximo para la velocidad a la que se pueden ir dando los pasos, pero ofrecen mayor torque. De hecho, cuando están parados (pero siguen recibiendo alimentación eléctrica), los steppers ofrecen un torque muy elevado porque sus bobinas internas ejercen de freno, con lo que resulta muy difícil mover su eje. Existen dos tipos de motores paso a paso:

Unipolares: tienen cuatro bobinas internas (en realidad, no son cuatro bobinas: son solo dos, cada una de las cuales está dividida por una conexión central común por donde recibe la alimentación) rodeando el eje central, el cual está imantado. Del motor salen seis cables: cuatro de ellos corresponden a los extremos de las dos bobinas y los otros dos a sus respectivas conexiones centrales comunes. Puede haber modelos de motor que solo ofrezcan cinco cables: en ese caso, una de las conexiones centrales se conecta internamente a la otra y ambas se alimentan por tanto a partir de un único cable.

Para saber qué cable es cada cual, se debería de consultar el datasheet, aunque hay un método "manual" para averiguarlo: medir con un multímetro la resistencia existente entre los extremos de dos de los cables. La resistencia entre los cables extremos de cada bobina es el doble que la existente entre uno de esos cables y el de la conexión central. Y la resistencia entre cables pertenecientes a bobinas diferentes es infinita. Sabiendo esto, la averiguación es sencilla.

Bipolares: tienen dos bobinas internas independientes (que no están divididas por ninguna conexión central) rodeando el eje central, el cual está imantado. Del motor salen cuatro cables, que corresponden a los extremos de cada bobina. Los motores bipolares tienen aproximadamente un 30% más de torque que un motor unipolar del mismo volumen, pero su circuitería es algo más compleja.

El control del movimiento de los motores paso a paso se realiza aplicando cierto voltaje a sus bobinas (a través de los cuatro cables conectados en sus extremos respectivos), el cual ha de seguir un determinado patrón repetitivo. Según van recibiendo tensión o dejándola de recibir en cada paso de ese patrón, las bobinas crean los campos magnéticos adecuados para atraer o repeler los imanes del eje, causando así su rotación. Por tanto, mediante el patrón de corriente adecuado, podemos controlar el movimiento del motor al detalle.

Los patrones pueden ser muy variados. Un patrón típico podría ser, por ejemplo, el mostrado en la siguiente tabla. En cada paso de este patrón se activan dos bobinas, lo que provoca que el eje se quede situado entre ellas (las letras A, B, C y D representan las cuatro bobinas del motor). Después del paso 4 se volvería a empezar por el 1, y así. Es fácil deducir que invirtiendo el orden de un patrón, invertimos el sentido del movimiento.

PASO	BOBINA A	BOBINA B	BOBINA C	BOBINA D	GIRO
1	**ON**	**ON**	OFF	OFF	
2	OFF	**ON**	**ON**	OFF	
3	OFF	OFF	**ON**	**ON**	
4	**ON**	OFF	OFF	**ON**	

Otro patrón bastante utilizado (aunque con menor torque de paso y de paro) es activar en cada paso una sola bobina; de esta forma, el eje se encararía directamente a ella en vez de quedar en medio de dos. También se pueden combinar ambos para realizar un patrón de 8 pasos: A, A-B, B, B-C, C, C-D, D, D-A, etc.

La circuitería necesaria para poder enviar los patrones de señal correctos a las bobinas de un motor unipolar o bipolar (y así poderlos controlar) es relativamente compleja. En el caso de los motores bipolares, es necesario conectar a nuestro microcontrolador un “puente H” por cada bobina del motor, así que en realidad siempre necesitaremos dos “puentes H” iguales. En el caso de los motores

unipolares, lo más habitual es utilizar un conjunto de 8 transistores de tipo Darlington, normalmente encapsulados dentro de un mismo chip, como por ejemplo el ULN2003: conectando de la forma adecuada las patillas del ULN2003 a nuestro microcontrolador, las cuatro bobinas podrán ser directamente gestionadas por este.

Nota: un transistor Darlington es un tipo de transistor que está compuesto internamente por dos transistores (normalmente NPN) conectados en cascada, por lo que tiene una alta ganancia de corriente debida a la multiplicación de la ganancia de los dos transistores bipolares. Que estén conectados en cascada significa que el primer transistor entrega la corriente que sale por su emisor a la base del segundo transistor.

Como la mayoría de motores, los steppers requieren más electricidad de la que la placa Arduino les puede ofrecer, así que necesitaremos una fuente externa para alimentarlos. Existen muchos modelos de steppers, y cada uno de ellos trabaja a un determinado voltaje DC; para saber cuál es el voltaje de trabajo de nuestro motor deberíamos consultar el datasheet ofrecido por el fabricante. Valores típicos son 5 V, 9 V, 12 V y 24 V.

Adafruit distribuye los siguientes steppers: el producto nº 858 (que funciona a 5V), el nº 324 y el nº 918 (que funcionan ambos a 12 V). Sparkfun por su parte también distribuye diferentes steppers, como los nº 10551, 9238, 10846, 10847 o 10848, entre otros. En las especificaciones técnicas de cada motor se indica el número de pasos (N_p) que puede realizar como máximo en una vuelta, pero dependiendo del motor, hay que tener en cuenta que para obtener el número real de pasos posibles, el fabricante ha de indicar si N_p ha de ser multiplicado por el factor de reducción del tren de engranajes. Es decir, un motor de N_p=48 pasos que tenga 1/16 de factor de reducción en realidad podrá hacer 48·16=768 pasos por vuelta.

La librería Servo

Aunque podríamos controlar un servomotor "a pico y pala" generando directamente una señal digital de 50 Hz con pulsos cuya duración variara entre 1 ms y 2 ms, es mucho más sencillo (y conveniente) utilizar la librería oficial "Servo". Lo primero que hay que hacer para poder controlar un servomotor utilizando esta librería es declarar un objeto de tipo Servo. Si suponemos que lo llamamos "miservo", esto se haría con la línea: `Servo miservo;` en la zona de declaraciones globales. A partir de aquí, la primera función imprescindible que debemos escribir (normalmente dentro de "setup()") es:

miservo.attach(): vincula el objeto “miservo” con el pin digital de la placa Arduino donde está conectado físicamente el cable de control del servomotor. Su parámetro precisamente es el número de ese pin. Esta función no tiene valor de retorno.

A partir de aquí, ya podemos hacer uso del resto de funciones de la librería Servo para controlar los servomotores de nuestro circuito:

miservo.write(): controla el eje del servomotor. Su único parámetro es el ángulo (en grados) donde se quiere situar dicho eje. Como resultado de ejecutar esta instrucción en un servomotor estándar, se obtendrá un movimiento del eje hasta alcanzar ese ángulo especificado. En un servomotor de rotación continua, en cambio, el valor de su parámetro representa la velocidad del eje: 0 equivale a la máxima velocidad en un sentido, 180 a la máxima velocidad en el sentido contrario y 90 equivale aproximadamente a inexistencia de movimiento. Esta función no tiene valor de retorno.

miservo.writeMicroseconds(): controla el eje del servomotor. Su parámetro –de tipo “int”–, en vez de ser el ángulo en grados como en *miservo.write()*, es la duración del pulso de la señal de control. Un valor de 1500 mantiene el eje en la posición central, un valor de 1000 lo mueve hasta su posición mínima y un valor de 2000 lo mueve hasta el otro extremo. Algunos fabricantes no siguen este estándar muy estrictamente, por lo que los servomotores a menudo responden a valores entre 700 y 2300, así que en la práctica deberemos ir probando de aumentar los límites hasta que el servomotor no aumente su movimiento. De todas formas, intentar hacer funcionar un servomotor más allá de sus límites (frecuentemente notificado por un ruido característico) es un estado de alta corriente y debería ser evitado. Los motores de rotación continua responden a esta función de la misma manera que con *miservo.write()*: variando su velocidad de rotación entre un mínimo y un máximo por cada sentido. Esta función no tiene valor de retorno.

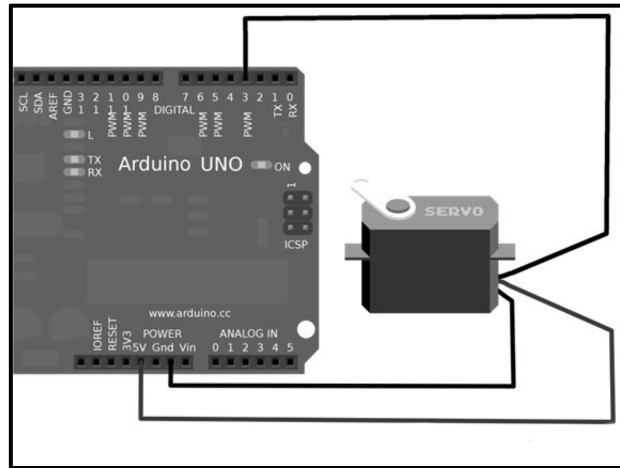
miservo.read(): devuelve el ángulo actual del servomotor (es decir, el valor pasado en la última ejecución de *miservo.write()*).No tiene ningún parámetro.

miservo.attached(): comprueba si el objeto “miservo” está vinculado a algún pin de la placa Arduino. Si es así, devuelve “true”, si no, “false”. No tiene ningún parámetro.

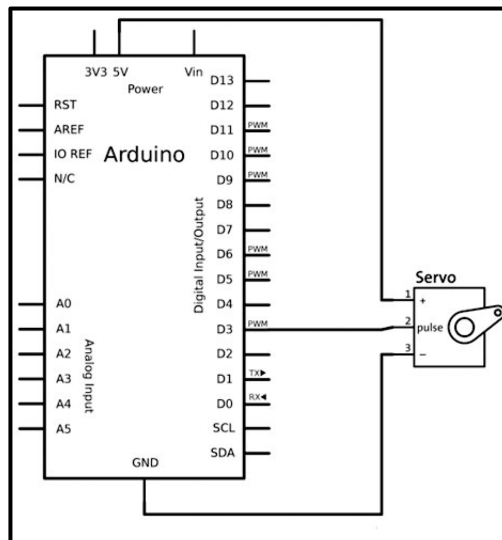
miservo.detach(): desvincula el objeto “miservo” del pin de la placa Arduino asociado. Solamente cuando todos los objetos Servo existentes en nuestro

sketch sean desvinculados, los pines 9 y 10 de la placa podrán ser utilizados como salida PWM estándar otra vez. No tiene ni parámetros ni valor de retorno.

Veamos unos ejemplos de uso. Conectemos un servo a nuestra placa Arduino tal como se muestra a continuación (nota: el pin digital donde se conecta el cable de control del servomotor NO tiene por qué ser de tipo PWM):



El diseño anterior tendría un esquema eléctrico tal como el siguiente:



Ejemplo 5.15: El siguiente código mueve de forma continua el eje del servomotor desde un ángulo de 0 grados hasta 180 y seguidamente lo mueve en sentido contrario de 180 grados a 0, y así ininterrumpidamente.

```
#include <Servo.h>
Servo miservo;
int i = 0;
void setup() {
       miservo.attach(3);
}
void loop(){
      //Va de 0 a 180 grados en pasos de un grado
      for(i = 0; i < 180; i++)    {
                 miservo.write(i);
                 //Esperamos a que el servo alcance la nueva posición
                 delay(15);
      }
      //De 180 a 0
      for(i = 180; i>=1; i--) {
                 miservo.write(i);
                 delay(15);
      }
}
```

Ejemplo 5.16: Otro ejemplo puede ser (con el mismo circuito anterior) ordenar al servomotor que se sitúe en un ángulo determinado especificado a través del canal serie:

```
#include <Servo.h>
Servo miservo;
void setup() {
       Serial.begin(9600);
       miservo.attach(3);
}
void loop(){
       long angulo;
       if (Serial.available() > 0) {
            angulo=Serial.parseInt();
            if (angulo > 0 && angulo < 180){
                miservo.write(angulo);
                delay(15);
            }
       }
}
```


Ejemplo 5.17: Supondremos que tenemos dos servomotores situados frente a frente, con una rueda anclada a cada servomotor. Con el siguiente código podremos mover las dos ruedas en el mismo sentido (adelante o atrás) o bien en sentidos contrarios (provocando así giros hacia la derecha o hacia la izquierda).

```
#include <Servo.h>
Servo servoIzq;
Servo servoDer;
void setup() {
  servoIzq.attach(10);
  servoDer.attach(9);
}
void loop() {
  adelante();      delay(2000);
  atras();         delay(2000);
  giroDerecha();   delay(2000);
  giroIzquierda(); delay(2000);
  parar();         delay(2000);
}
void adelante()      { servoIzq.write(0);    servoDer.write(180); }
void atras()         { servoIzq.write(180);  servoDer.write(0);   }
void giroDerecha()   { servoIzq.write(180);  servoDer.write(180); }
void giroIzquierda() { servoIzq.write(0);    servoDer.write(0);   }
void parar()         { servoIzq.write(90);   servoDer.write(90);  }
```

Respecto a las conexiones a realizar para poner en funcionamiento el ejemplo anterior en la vida real, hay que tener en cuenta que alimentar dos servomotores por el pin de 5 V de la placa Arduino puede ser un poco justo por el consumo que estos realizan, así que lo recomendable es conectar el cable de alimentación de los servomotores al terminal positivo de una fuente externa (como por ejemplo una pila de 9 V), y el cable de tierra de los servomotores al terminal negativo de dicha fuente. Al hacer esto, es muy importante no olvidarse de unir las dos tierras existentes en el circuito: la de la fuente externa (representada por su terminal negativo) y la de la placa Arduino (tierra que en general es diferente de la primera porque la placa se suele alimentar de una fuente diferente: el cable USB); si no compartimos las dos tierras para que sean una sola, el comportamiento del circuito puede ser muy errático. Por otro lado, lógicamente cada servomotor deberá conectarse además a un pin digital de la placa Arduino (en el sketch son los nº 9 y los nº 10).

Veremos más ejemplos de uso de servomotores en el siguiente capítulo, cuando estudiemos las entradas y salidas del microcontrolador.

La librería Stepper

Los motores paso a paso, tal como se han comentado, requieren el uso de un conjunto de transistores Darlington (si son unipolares) o de dos “puentes H” (si son bipolares). En el primer caso, se pueden utilizar chips integrados como el ULN2003 o el ULN2004 (la diferencia entre ambos está en el valor de la resistencia de sus entradas) o también el ULN2803 o el ULN2804 (con una salida más que los anteriores), o similares. En el segundo caso, se pueden utilizar chips integrados como el L293, el L298 o el SN754410, los cuales admiten diferentes grados de corriente de salida.

No obstante, como el diseño de circuitos usando estos chips integrados es relativamente complejo y varía según el modelo escogido de chip (ya que cada uno tiene una disposición de pines diferentes), en este libro no implementaremos “a mano” ningún circuito de control de steppers, y optaremos por utilizar shields o módulos específicos que ya tengan incorporados toda esta circuitería necesaria. De esta manera, tan solo nos tendremos que preocupar de conectar convenientemente los cables del motor paso a paso (4 o 5/6 según si es bipolar o unipolar, respectivamente) a los zócalos adecuados del shield/módulo y de alimentarlo: recordemos que los motores paso a paso necesitan alimentación externa (con 9 V o 12 V normalmente ya es suficiente, pero dependerá del modelo) porque los 5 V que ofrece la placa Arduino se quedan cortos.

Si el lector aún desea construir un circuito de control de motores paso a paso desde cero, comentaremos simplemente que para el caso de un motor paso a paso unipolar podemos consultar dos diagramas disponibles en la página oficial de Arduino (http://arduino.cc/en/Reference/StepperUnipolarCircuit). En ambos diagramas interviene el chip ULN2004, pero en uno se utilizan dos pines-hembra de la placa Arduino como salidas digitales (uno por cada bobina) y en el otro, más sencillo, se utilizan cuatro (uno por cada extremo de cada bobina). Para el caso de un motor paso a paso bipolar, podemos consultar otros dos diagramas en http://arduino.cc/en/Reference/StepperBipolarCircuit. En ambos diagramas aparece el chip L293D (o equivalente), pero en uno se utilizan dos pines de la placa Arduino y en otro se utilizan cuatro.

Si nos decidimos a usar el Motor Shield oficial de Arduino, una vez hechas las conexiones pertinentes, es posible controlar un motor paso a paso sin necesidad de utilizar ninguna librería específica.

Ejemplo 5.18: Como ejemplo de ello, a continuación se muestra un código que realiza una secuencia simple de 4 pasos, concretamente el patrón de tipo A-B-C-D.


```
int motorPin1 = 8;   //Pin conectado a un extremo de la bobina 1
int motorPin2 = 9;   //Pin conectado al otro extremo de la bobina 1
int motorPin3 = 10;  //Pin conectado a un extremo de la bobina 2
int motorPin4 = 11;  //Pin conectado al otro extremo de la bobina 2
int delayTime = 500; //Tiempo que determina la velocidad de giro
void setup(){
     pinMode(motorPin1, OUTPUT);
     pinMode(motorPin2, OUTPUT);
     pinMode(motorPin3, OUTPUT);
     pinMode(motorPin4, OUTPUT);
}
void loop(){
     digitalWrite(motorPin1, HIGH); // Primer paso
     digitalWrite(motorPin2, LOW);
     digitalWrite(motorPin3, LOW);
     digitalWrite(motorPin4, LOW);
     delay(delayTime);
     digitalWrite(motorPin1, LOW); // Segundo paso
     digitalWrite(motorPin2, HIGH);
     digitalWrite(motorPin3, LOW);
     digitalWrite(motorPin4, LOW);
     delay(delayTime);
     digitalWrite(motorPin1, LOW); // Tercer paso
     digitalWrite(motorPin2, LOW);
     digitalWrite(motorPin3, HIGH);
     digitalWrite(motorPin4, LOW);
     delay(delayTime);
     digitalWrite(motorPin1, LOW); // Cuarto paso
     digitalWrite(motorPin2, LOW);
     digitalWrite(motorPin3, LOW);
     digitalWrite(motorPin4, HIGH);
     delay(delayTime);
}
```

No obstante, la librería oficial Stepper nos ofrece la posibilidad de controlar un motor paso a paso de una forma más sencilla. De hecho, solo tiene tres funciones:

Stepper(): devuelve un objeto de tipo Stepper (lo llamaremos "mistepper") que representa un determinado motor paso a paso conectado a la placa Arduino. Este objeto lo utilizaremos en nuestro sketch para controlar dicho motor. La función *Stepper()* tiene la particularidad de que ha de ser escrita al principio del código de nuestro sketch (en la zona de declaraciones de variables globales, fuera por tanto incluso de "setup()"). Tiene varios

parámetros: el valor del primero –de tipo “int”– es el número de pasos en una vuelta que es capaz de dar el motor como máximo. Este dato nos lo tiene que ofrecer el fabricante. No obstante, a veces lo que nos ofrece es otro dato: el número de grados por paso; si es así, podemos obtener fácilmente el número de pasos totales si dividimos 360 entre los grados por paso (es decir, si un motor tiene por ejemplo 3,6 grados por paso, es de 360/3,6=100 pasos). El segundo y tercer parámetro –de tipo “int” también– representan los dos pines digitales de la placa Arduino donde se conectará el motor si el diseño del circuito es con solo dos pines; si el diseño requiriera cuatro pines, se debería utilizar un cuarto y quinto parámetro extra que representan esos otros dos pines más de la placa Arduino.

mistepper.setSpeed(): establece la velocidad del motor en vueltas por minuto (rpm). Esta velocidad se especifica como parámetro, el cual, aunque es de tipo “long”, ha de ser positivo. Esta función no hace que el motor gire: solamente especifica su velocidad en el momento que este empiece a girar (gracias a la ejecución de *mistepper.step()*). No tiene valor de retorno.

mistepper.step(): gira el motor un número determinado de pasos, especificado como parámetro –de tipo “int”–. Si este número es positivo, girará en un sentido, y si es negativo, en el otro. La velocidad de giro viene determinada por la última ejecución de *mistepper.setSpeed()*. Atención: esta función es bloqueante. Esto quiere decir que el sketch se esperará hasta que el motor haya acabado de moverse para continuar ejecutándose. Por lo tanto, si por ejemplo a un motor de 100 pasos le hacemos girar 100 vueltas a 1 rpm, *mistepper.step()* tardará un minuto completo en acabar, bloqueando mientras tanto el resto del sketch. Para un mejor control, por tanto, se recomienda mantener una velocidad elevada y pocos pasos. Esta función no tiene valor de retorno.

Ejemplo 5.19: Suponiendo que tenemos un motor paso a paso (unipolar o bipolar, de igual) conectado a nuestra placa Arduino (usando dos o cuatro cables), podemos ejecutar el siguiente sketch para observar su giro paso a paso lentamente en una dirección:

```
#include <Stepper.h>
/*El valor del primer parámetro se tendrá que cambiar según el modelo
de motor usado. Si se utiliza un circuito con solo dos pines, el
cuarto y quinto parámetro no se han de escribir. */
Stepper miStepper(200, 8,9,10,11);
```



```
int contador = 0;
void setup() {}
void loop() {
      miStepper.step(1);
      delay(500);
}
```

Ejemplo 5.20: El siguiente código, en cambio, hace girar el stepper una revolución entera en un sentido y otra en el otro:

```
#include <Stepper.h>
//Suponemos un motor de 200 pasos y un circuito de 4 pines
Stepper miStepper(200, 8,9,10,11);
void setup() {
     miStepper.setSpeed(60);  /*60 vueltas/minuto */
}
void loop() {
     //Realizo todos los pasos que hay en una vuelta entera
     miStepper.step(200);
     delay(500);
     //Ahora en sentido contrario
     miStepper.step(-200);
     delay(500);
}
```

También es posible cambiar la velocidad de giro del stepper (mediante *miStepper.setSpeed();*) especificando como parámetro un valor variable, obtenido por ejemplo de una lectura de algún sensor (lo veremos en el próximo capítulo).

En la wiki oficial de Arduino hay una librería alternativa para el control de motores paso a paso, algo más completa. Se llama "CustomStepper" y puede obtener de aquí: http://arduino.cc/playground/Main/CustomStepper . Otra librería alternativa, que también aporta más control que la oficial es la "AccelStepper" (http://www.open.com.au/mikem/arduino/AccelStepper). Ambas librerías permiten acelerar/desacelerar el movimiento del motor, la gestión independiente de múltiples motores, funciones no bloqueantes, mejor soporte a otros chips controladores, etc.

6 ENTRADAS Y SALIDAS

La utilidad más evidente de una placa Arduino es interaccionar con su entorno físico a través de sensores y actuadores. Para ello, disponemos de varias funciones que tratan señales de tipo digital (ya sean entradas o salidas) o de tipo analógico (ya sean entradas o salidas).

USO DE LAS ENTRADAS Y SALIDAS DIGITALES

Las funciones que nos ofrece el lenguaje Arduino para trabajar con entradas y salidas digitales son:

pinMode(): configura un pin digital (cuyo número se ha de especificar como primer parámetro) como entrada o como salida de corriente, según si el valor de su segundo parámetro es la constante predefinida INPUT o bien OUTPUT, respectivamente. Esta función es necesaria porque los pines digitales a priori pueden actuar como entrada o salida, pero en nuestro sketch hay que definir previamente si queremos que actúen de una forma u de otra. Es por ello que esta función se suele escribir dentro de "setup()". No tiene valor de retorno.

Si el pin digital se quiere usar como entrada, es posible activar una resistencia “pull-up” de 20 KΩ que todo pin digital incorpora. Para ello, se ha de utilizar la constante predefinida INPUT_PULLUP en vez de INPUT, ya que la constante INPUT desactiva explícitamente estas resistencias “pull-ups” internas. Recordemos que si un pin de entrada tiene su resistencia “pull-up” interna desactivada, en el momento que no esté conectado a nada puede recibir ruido eléctrico del entorno o de algún pin cercano y provocar así inconsistencias en los valores de entrada obtenidos (ya que estos cambiarán aleatoriamente en cualquier momento). Esto hace que por lo general sea recomendable activar la resistencia “pull-up”.

Otra forma alternativa de activar la resistencia “pull-up”, diferente de la descrita en el párrafo anterior, es utilizar la constante INPUT y además la función *digitalWrite()* de una forma muy concreta –ver siguiente párrafo–. Otra forma diferente de conseguir el mismo resultado práctico sería utilizar en vez de la resistencia “pull-up” interna, una resistencia “pull-down” extra externa (es decir, una resistencia que conectara ese pin a tierra); un valor de 10 KΩ ya valdría.

digitalWrite(): envía un valor ALTO (HIGH) o BAJO (LOW) a un pin digital; es decir, tan solo es capaz de enviar dos valores posibles. Por eso, de hecho, hablamos de salida “digital”. El pin al que se le envía la señal se especifica como primer parámetro (escribiendo su número) y el valor concreto de esta señal se especifica como segundo parámetro (escribiendo la constante predefinida HIGH o bien la constante predefinida LOW, ambas de tipo “int”).

Si el pin especificado en *digitalWrite()* está configurado como salida, la constante HIGH equivale a una señal de salida de hasta 40 mA y de 5 V (o bien 3,3 V en las placas que trabajen a ese voltaje) y la constante LOW equivale a una señal de salida de 0 V. Si el pin está configurado como entrada usando la constante INPUT, enviar un valor HIGH equivale a activar la resistencia interna “pull-up” en ese momento (es decir, es idéntico a usar directamente la constante INPUT_PULLUP), y enviar un valor LOW equivale a desactivarla de nuevo. Esta función no tiene valor de retorno.

Es posible que el valor HIGH (5V) ofrecido por un pin digital de salida no sea suficiente para alimentar componentes de alto consumo (como motores o matrices de LEDs, entre otros), por lo que estos deberían alimentarse siempre con circuitería extra (evitando así además posibles daños en el pin por sobrecalentamiento). O por el contrario, que dicho valor se deba reducir mediante un divisor de tensión para

adecuarlo al voltaje de trabajo del componente allí conectado. En las páginas siguientes se irán viendo ejemplos de uno y de lo otro.

digitalRead(): devuelve el valor leído del pin digital (configurado como entrada mediante *pinMode()*) cuyo número se haya especificado como parámetro. Este valor de retorno es de tipo "int" y puede tener dos únicos valores (por eso, de hecho hablamos de entrada digital): la constante HIGH (1) o LOW (0).

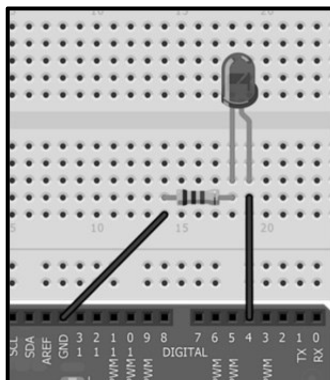
Si la entrada es de tipo INPUT, el valor HIGH se corresponde con una señal de entrada mayor de 3 V y el valor LOW con una señal menor de 2 V. Si la entrada es de tipo INPUT_PULLUP, al tener la entrada conectada una resistencia "pull-up", las lecturas son al revés: el valor HIGH indica que no se recibe señal de entrada y el valor LOW que sí.

Además de las anteriores, otra función no tan usada pero interesante es:

pulseIn(): pausa la ejecución del sketch y se espera a recibir en el pin de entrada especificado como primer parámetro la próxima señal de tipo HIGH o LOW (según lo que se haya indicado como segundo parámetro). Una vez recibida esa señal, empieza a contar los microsegundos que esta dura hasta cambiar su estado otra vez, y devuelve finalmente un valor –de tipo "long"– correspondiente a la duración en microsegundos de ese pulso de señal. De forma opcional, se puede especificar un tercer parámetro –de tipo "unsigned long"– que representa el tiempo máximo de espera en microsegundos: si la señal esperada no se produce una vez superado este tiempo de espera, la función devolverá 0 y continuará la ejecución del sketch. Si este tiempo de espera no se especifica, el valor por defecto es de un segundo. En la documentación oficial recomiendan usar esta función para rangos de valores de retorno de entre 10 microsegundos y 3 minutos, ya que para pulsos más largos la precisión puede tener errores.

Ejemplos con salidas digitales

Ejemplo 6.1: Veamos un código muy sencillo para ilustrar el uso de las funciones *pinMode()* y *digitalWrite()*. La idea es encender y apagar un LED de forma periódica, simplemente. El circuito montado en la breadboard sería parecido a este:



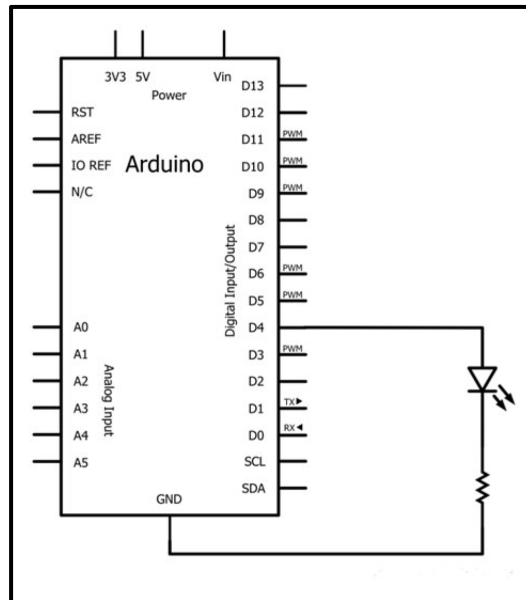
En él se puede observar que el terminal positivo del LED está conectado al pin-hembra digital número 4 de la placa Arduino y su terminal negativo está conectado a una resistencia que hace de divisor de tensión (un valor de 220 ohmios ya estaría bien), la cual está conectada a su vez a tierra.

Evidentemente, para que este circuito funcione la placa Arduino ha de recibir alimentación eléctrica de alguna manera (en la figura anterior no se muestra la fuente porque para el caso es irrelevante). Una vez la placa esté encendida (y por tanto, nuestro sketch –mostrado en la página siguiente– esté ejecutándose), cuando así lo decida nuestro programa la placa enviará corriente al LED a través de su pin 4 configurado como salida (y por tanto, lo encenderá), y cuando toque dejará de enviarle corriente (y por tanto, lo apagará).

Notar que no se ha tenido que hacer uso de los pines-hembra “5V” o “Vin” de la placa Arduino; esto es un hecho que confunde mucho a los principiantes. Los pines-hembra de alimentación solamente son necesarios cuando queremos aportar electricidad de forma permanente y estable a algún componente de nuestro circuito (como un LED, por ejemplo) independientemente del estado de las salidas digitales. Tal como veremos en el código de ejemplo, para encender el LED enviamos un voltaje de 5 V (el valor HIGH) a la salida digital en la que está conectado porque queremos tenerlo encendido solo cuando se envía precisamente ese valor HIGH, no todo el tiempo. Eso sí, los valores HIGH que puede ofrecer la placa Arduino a través de sus pines de salida son gracias a que es ella la que se alimenta.

En cualquier caso, lo que sí que ha de tener todo circuito es una toma de tierra. Por tanto, en todos nuestros proyectos con Arduino siempre utilizaremos algún pin-hembra de los que están marcados como “GND”.

El esquema correspondiente al circuito anterior será este:



Y he aquí finalmente el código:

```
void setup(){
	//Inicializamos el pin digital 4 como salida.
	//Es donde irá conectado nuestro LED.
	pinMode(4,OUTPUT);
}
void loop(){
	digitalWrite(4,HIGH); //Decimos que se encienda el LED
	delay(1000);          //Esperamos un segundo
	digitalWrite(4,LOW);  //Decimos que se apague el LED
	delay(1000);          //Esperamos un segundo
}
```

El sketch anterior es bastante autoexplicativo: tras establecer el pin digital número 4 como salida, enviamos gracias a la función *digitalWrite()* una señal de valor HIGH (5V) a través de ella. Si no se hiciera nada en nuestro código para evitarlo, esta señal, una vez activada, continuaría enviándose sin interrupción. Pero como lo que queremos es hacer parpadear el LED, justo después de enviar la señal de valor HIGH,

enviamos una señal de valor LOW (0V) para así interrumpir el encendido. Una vez llegados al final del "loop", volvemos para arriba y continuamos el proceso: encendido-apagado-->encendido-apagado… y así.

¿Por qué intercalamos las funciones *delay()* en medio de las dos *digitalWrite()*? Porque si no estuvieran, el envío de la señal HIGH y el de la señal LOW se harían sumamente seguidos (ya que la velocidad con la que el microcontrolador pasa de ejecutar una línea del sketch a la siguiente es tremendamente alta; de hecho, eventos escritos en líneas contiguas dentro de la función "loop()" los podemos considerar prácticamente simultáneos) y nuestro ojo no podría distinguir ningún parpadeo. La función *delay()* sirve para pausar ("congelar") el sketch un determinado tiempo y así mantener la señal previamente enviada por los dos *digitalWrite()* de forma que se pueda apreciar. Es evidente, pues, que escribir un tiempo menor en *delay()* hará que el parpadeo sea más rápido, y un tiempo mayor lo contrario. De hecho, si reducimos el parámetro de *delay()* hasta aproximadamente 10 milisegundos, veremos que efectivamente el LED deja de parpadear; esto es porque el parpadeo es tan rápido que nuestro ojo ya es incapaz de observarlo. Como dato curioso, si con 10 ms de *delay()* además movemos nuestro montaje de un lado a otro dentro de una habitación oscura, veremos que el LED deja un camino de luz.

Teniendo lo anterior en cuenta, podríamos jugar un poco con el código. Por ejemplo, si cambiáramos el tiempo de espera de uno de los *delay()* podríamos mantener encendido o apagado el LED más tiempo o menos. Sabiendo esto (y añadiendo algún *digitalWrite()* y *delay()* más), podríamos realizar un ejercicio interesante: mostrar una secuencia de encendidos/apagados correspondientes a diferentes códigos Morse (tal como el S.0.S., compuesto por tres puntos, tres rayas y tres puntos), donde la raya equivaldría a un "encendido largo", el punto a un "encendido corto" y la pausa entre ellos a un apagado de duración única.

<u>Ejemplo 6.2</u>: Veamos otro ejemplo de salidas digitales (seguiremos utilizando LEDs porque su comportamiento es fácilmente observable). El circuito de este nuevo ejemplo es exactamente igual al utilizado en el ejemplo anterior, y su comportamiento también (es decir, se consigue el parpadeo periódico de un LED). La novedad está en el código de nuestro sketch, donde para generar el tiempo de espera entre los estados encendido y apagado del LED no se utiliza ninguna función *delay()*. ¿Qué aporta este cambio? Que podamos ejecutar instrucciones de nuestro código al mismo tiempo que se está encendiendo o apagando el LED. Es decir: con *delay()* todo sketch se pausa hasta que no termina el tiempo especificado y continúa su ejecución; si por ejemplo queremos hacer parpadear un LED mientras queremos a la vez estar

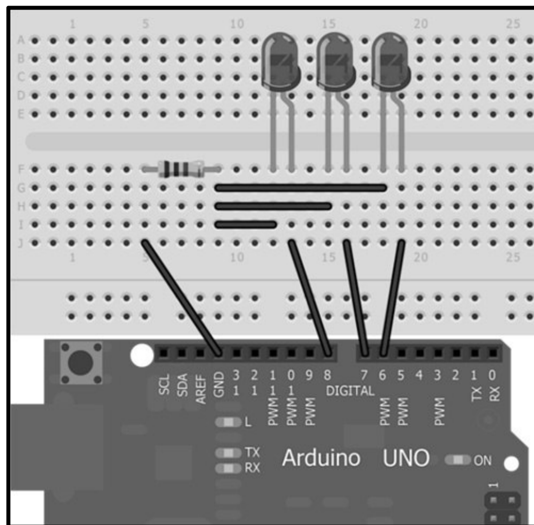
pendientes de la llegada de algún dato por algún pin de entrada o por el canal serie, no podemos usar *delay()* porque nuestro programa al llegar a la línea de esta instrucción se pararía completamente y se estaría "perdiendo" lo ocurrido "ahí fuera".

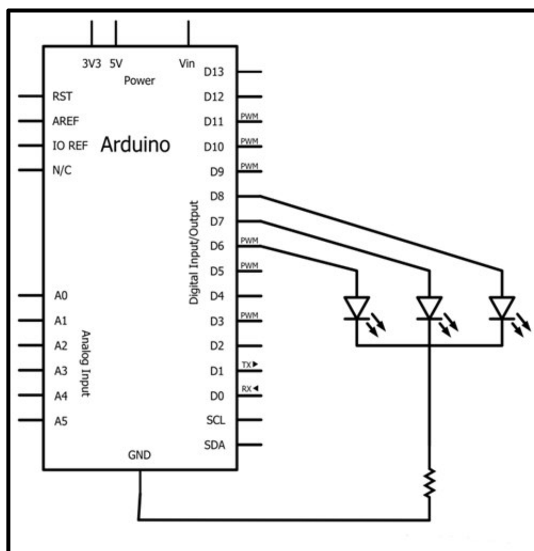
El truco para no usar *delay()*, tal como se puede ver en el código siguiente, es guardar el tiempo de la última vez que el LED se encendió o se apagó y comprobar entonces en cada vuelta del "loop()" si ya ha pasado el tiempo suficiente para cambiar el estado del LED. Evidentemente, esta idea se puede generalizar para gestionar la periodicidad de todo tipo de eventos.

```
int estadoLed = LOW; //Variable para cambiar el estado del LED
long t1 = 0 ; //Guarda el último momento cuando el LED cambió
long t2 = 0 ; //Guarda el instante actual de ejecución
long intervalo = 1000; //Intervalo de tiempo para el parpadeo
void setup(){
	pinMode(4,OUTPUT);
}
void loop() {
/*Aquí viene el  código que se desea ejecutar todo el rato (lectura
de sensores, control de actuadores, etc.) */

/*Ahora se pasa a comprobar si ha de cambiar el estado del LED. Esto
se hace mirando si la diferencia entre el tiempo actual y la última
vez que el LED cambió es mayor que el intervalo que se quiere para el
parpadeo.*/
	t2=millis();
	if (t2 – t1 > intervalo){
		//Guardo el tiempo del cambio, para la próxima vez
		t1 = t2;
		//Y realizo el cambio de estado
		if (estadoLed == LOW){
			estadoLed = HIGH;
		} else {
			estadoLed = LOW;
		}
		digitalWrite(4, estadoLed);
	}
}
```

Ejemplo 6.3: Veamos otro ejemplo más de salidas digitales. Como se puede ver en las figuras siguientes, ahora tenemos 3 LEDs conectados en paralelo. Cada uno de ellos recibirá una señal digital proveniente de un pin-hembra de la placa Arduino (en nuestro ejemplo, son el 6, 7 y 8 respectivamente, pero podrían ser otros cualesquiera). La tarea del circuito es realizar un encendido de los LEDs de tal forma que simule el efecto “coche fantástico” (sin estela); es decir, que se enciendan los LEDS por este orden: 6, 7, 8, 7, 6, 7, 8…





Fijarse que como divisor de tensión hemos utilizado una sola resistencia. Podríamos haber utilizado una resistencia conectada en serie a cada LED (es decir, tres resistencias en total) para conseguir el mismo objetivo, pero tal como lo hemos hecho nos ahorramos dos resistencias. El código del sketch que la placa Arduino ha de ejecutar es este:

```
byte i=0;     //Variable contador para los bucles for
int del=100; //Variable que marca el tiempo en los delay()
void setup() {
/*Inicializamos los pines digitales 6, 7 y 8 como salida.
Hacerlo con un for nos ahorra muchas líneas de código */
	for (i = 6; i<=8 ; i++)    {
		pinMode(i, OUTPUT);
	}
}
void loop() {
      //Parpadean los LEDs del 6 al 8
	for (i = 6; i<=8; i++) {
		digitalWrite(i, HIGH);
		delay(del);
		digitalWrite(i, LOW);
	}
   /*Parpadean los LEDs del 8-1 (7) hasta el 6+1 (7). En
   este caso particular, no habría hecho falta ningún bucle
   porque solo hay un LED entre ambos extremos (el 7) pero
   se deja como demostración para posibles ampliaciones */
	for (i = 7; i>=7; i--) {
		digitalWrite(i, HIGH);
		delay(del);
		digitalWrite(i, LOW);
	}
}
```

Ejemplo 6.4: El siguiente es otro ejemplo de código que realiza lo mismo que el anterior, pero utilizando un array para almacenar los números de pines donde están conectados los LEDs. Además, introduce una “estela” al mantener iluminados a la vez un LED y el siguiente e ir avanzando.

```
//Este array tiene los nº de pines donde se conectan los LED
byte leds[]={6,7,8};
byte i=0;     //Variable contador para los bucles for
int del=30;  //Variable que marca el tiempo en los delay()
void setup() {
```

```
//El bucle recorre los tres elementos del array (0, 1 y 2)
	for (i=0;i<=2;i++) {
		pinMode(leds[i],OUTPUT);
	}
}
void loop() {
//Parpadean los LEDs, del 6 al 8
//Atención a los límites del bucle for
	for (i=0;i<2;i++) {
		digitalWrite (leds[i],HIGH);
		delay(del);
		digitalWrite(leds[i+1],HIGH);
		delay(del);
		digitalWrite (leds[i],LOW);
		delay(del*2);
	}
//Parpadean los LEDs, del 8 al 6
//Atención a los límites del bucle for
	for (i=2;i>0;i--) {
		digitalWrite (leds[i],HIGH);
		delay(del);
		digitalWrite(leds[i-1],HIGH);
		delay(del);
		digitalWrite (leds[i],LOW);
		delay(del*2);
	}
}
```

Se deja como ejercicio modificar el sketch anterior para que los tres LEDs se iluminen y se apaguen a la vez. ¡Es muy fácil!

Otro ejercicio interesante con tres LEDs en paralelo (si fueran de color verde, naranja y rojo sería perfecto) es escribir un sketch que simule el comportamiento de un semáforo. Es decir, hacer que el LED “verde” se encienda durante un rato, que al cabo de un tiempo empiece a parpadear el LED “naranja”, para seguidamente apagarse ambos y encenderse el LED “rojo”. Seguidamente se debería de volver a empezar el mismo proceso otra vez. Incluso con la ayuda de un pulsador (de cuyo uso hablaremos en el siguiente apartado) se podría obligar a encender el LED “verde” (y apagar los demás), tal como ocurre en algunos semáforos reales.

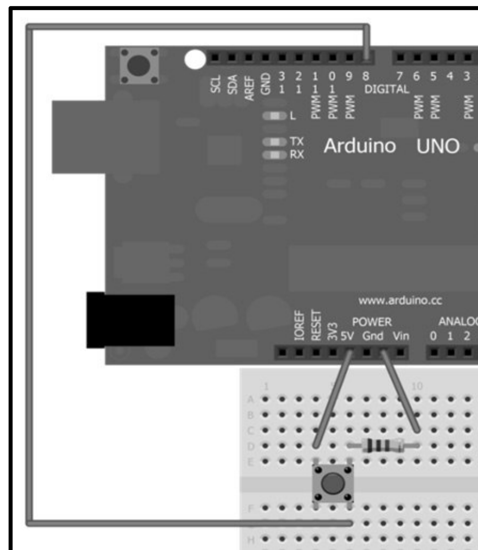
Ejemplo 6.5: Para acabar este apartado, mostraremos un sketch (para el mismo circuito de los tres LEDs) donde se ilustra el uso de funciones propias para reutilizar código. ¿Qué es lo que ocurriría si lo ponemos en marcha?

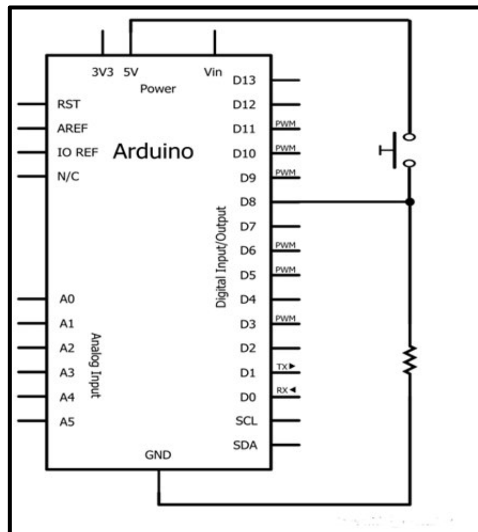
```
void setup(){
       pinMode(6, OUTPUT);
       pinMode(7, OUTPUT);
       pinMode(8, OUTPUT);
}
void loop() {
     /*Ojo: hasta que no una función no acaba de
      ejecutarse, el sketch no pasará a la siguiente */
       flash_led(6,500);
       flash_led(7,1000);
       flash_led(8,200);
}
void flash_led(byte unled, unsigned long espera)
{
      digitalWrite(unled, HIGH);
      delay(espera);
      digitalWrite(unled, LOW);
      delay(espera);
}
```

Ejemplos con entradas digitales (pulsadores)

En este apartado veremos varios ejemplos donde la placa Arduino realiza lecturas de señales digitales recibidas específicamente de parte de pulsadores. A priori esto puede parecer un poco extraño, ya que en principio un pulsador tan solo sirve para abrir o cerrar un circuito, pero hemos de saber que también tiene la capacidad de permitir la “monitorización” de su estado, de tal forma que con una señal de entrada digital la placa Arduino puede saber en todo momento si el pulsador está en posición abierta o posición cerrada. Para conseguir esta capacidad de “monitorización”, los pulsadores han de estar obligatoriamente conectados a una resistencia “pull-up” o bien “pull-down”. Así pues, tenemos dos posibilidades de circuitos.

Ejemplo 6.6: A continuación, mostramos el dibujo (y esquema correspondiente) del circuito que es capaz de monitorizar el estado de un pulsador utilizando una resistencia “pull-down”.





Podemos comprobar cómo las conexiones del pulsador son por un lado directa a la alimentación y por otro a tierra a través de la resistencia "pull-down" (pongamos que de 10 KΩ). Existe un tercer cable, conectado entre el pulsador y la resistencia que va a parar a un pin digital de la placa Arduino (en este caso particular, el nº 8). Este pin digital deberá configurarse como pin de entrada porque allí será donde se reciba la señal que indique el estado del pulsador.

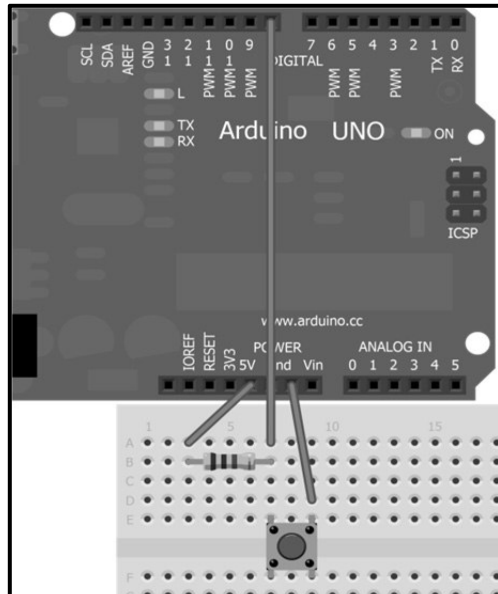
En esta configuración, cuando el botón está abierto (es decir, cuando no está pulsado), el “tercer cable” está conectado a tierra a través de la resistencia “pull-down”, por lo que recibe una señal de 0 V (LOW). Cuando el botón está cerrado (es decir, cuando sí está pulsado), el “tercer cable” se conecta al pin de alimentación, por lo que recibe una señal de 5 V (HIGH). Está claro que en este último caso, el “tercer cable” seguirá estando conectado a tierra, pero la corriente proveniente de la alimentación apenas se desviará a tierra por allí (porque precisamente la resistencia “pull-down” se opone a ello) y por tanto circulará a través del “tercer cable”, que ofrece una alternativa a los electrones para cerrar el circuito más fácilmente.

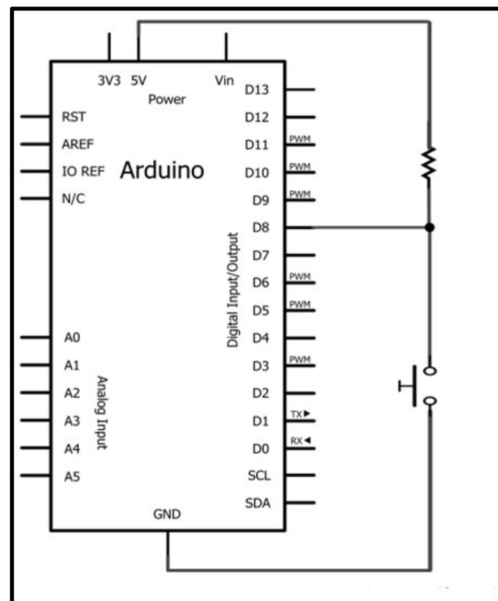
La necesidad de una resistencia “pull-down” se ve claramente si se desconecta el “tercer cable” de todo: la entrada empezará a “flotar” ya que no tiene ninguna conexión sólida a alimentación o tierra y los valores recibidos serán HIGH o LOW aleatoriamente.

Una vez diseñado el circuito, ¿cómo podemos observar el estado actual del pulsador en un sketch de Arduino? Una manera podría ser mediante el “Serial monitor”. A continuación, mostramos el código de ejemplo. Si lo ejecutamos, veremos que mientras tengamos pulsado el botón, en el “Serial monitor” aparecerá un 1, y cuando esté sin pulsar aparecerá un 0.

```
int estadoBoton=0; //Guardará el estado del botón (HIGH ó LOW)
void setup(){
	pinMode(8,INPUT); //Pin donde está conectado el pulsador
	Serial.begin(9600);
}
void loop() {
	estadoBoton=digitalRead(8);
	Serial.println(estadoBoton);
	delay (50);  //Para mayor estabilidad entre lecturas
}
```

Ejemplo 6.7: También podríamos monitorizar el estado de un botón usando una resistencia “pull-up”. En ese caso, el circuito y su esquema correspondiente serían así:





Aquí podemos comprobar cómo las conexiones del pulsador son por un lado a la alimentación a través de la resistencia “pull-up” (pongamos que de 10 KΩ) y por otro directa a tierra. Existe un tercer cable, conectado entre el pulsador y la resistencia que va a parar a un pin digital de la placa Arduino (en este caso particular,

el nº 8). Este pin digital deberá configurarse como pin de entrada porque allí será donde se reciba la señal que indique el estado del pulsador.

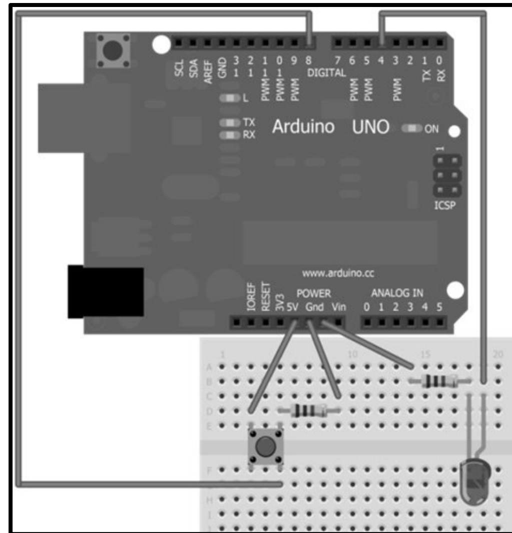
En esta configuración, cuando el botón está abierto, el "tercer cable" está conectado a la alimentación a través de la resistencia "pull-up", por lo que recibe una señal de 5 V (HIGH). Cuando el botón está cerrado, el "tercer cable" se conecta a tierra directamente, por lo que recibe una señal de 0 V (LOW). Está claro que en este último caso, el "tercer cable" seguirá estando conectado a la alimentación, pero la resistencia "pull-up" impide en la práctica el paso de electrones. Es importante notar, por tanto, que en esta configuración con la resistencia "pull-up", se recibe LOW cuando se pulsa el botón y HIGH cuando se deja de pulsar, al contrario de lo "convencional" y de lo que ocurre cuando se usan resistencias "pull-down".

El código que pusimos para monitorizar el pulsador en el circuito con resistencia pull-down sigue siendo válido ahora también con el circuito de resistencia "pull-up", pero veremos que los valores 0 y 1 están invertidos, tal como acabamos de comentar.

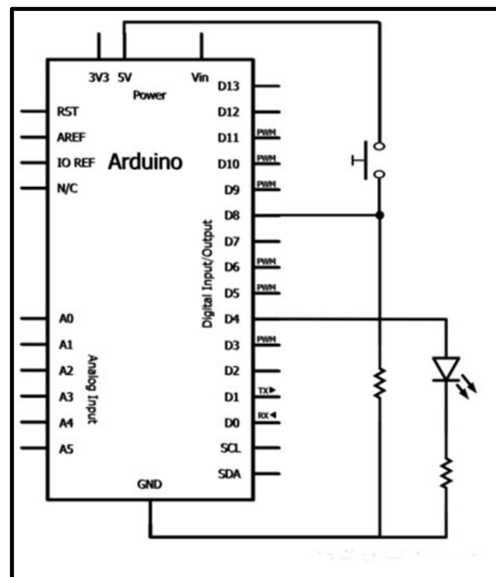
Recordemos que la placa Arduino tiene en cada uno de sus pines-hembra digitales una resistencia "pull-up" conectada internamente a la alimentación de 5 V. Esto quiere decir que si para un determinado pin-hembra la activáramos (utilizando en la función *pinMode();* la constante INPUT_PULLUP en vez de INPUT, recordemos), la conexión de un pulsador a ese pin-hembra sería mucho más sencilla. Concretamente deberíamos conectar un terminal del pulsador a tierra y el otro al pin-hembra deseado, y nada más. De todas formas, en esta configuración, seguiríamos recibiendo una señal HIGH al tener el pulsador abierto y una señal LOW al tenerlo cerrado.

Una vez conocidas las diferentes configuraciones posibles (con resistencias "pull-up" o "pull-down"), podemos empezar ya a diseñar circuitos que sean capaces de detectar el estado actual de un pulsador y reaccionar en consecuencia.

Ejemplo 6.8: Empezaremos por un caso muy claro y directo: el encendido de un LED mientras se mantiene pulsado un botón. Para ello en realidad lo que tenemos que hacer es diseñar dos circuitos independientes: uno para el manejo del pulsador y otro para el encendido del LED. El primero lo acabamos de estudiar (elegiremos el de la configuración "pull-down") y el segundo no es más que el primer circuito que vimos en los ejemplos de las salidas digitales. Los dos juntos tienen un aspecto como el mostrado en la siguiente figura:



En el dibujo anterior se puede observar que hay dos cables conectados a tierra, cada uno perteneciente a uno de los dos circuitos independientes. Esto se ha hecho así por claridad, pero es más habitual unir todos los cables a tierra físicamente en un solo cable final. En todo caso, la tierra siempre ha de ser común (cosa que en este caso está garantizado porque las diferentes tierras ofrecidas por la placa Arduino están todas conectadas entre sí, tal como se aprecia en el esquema del circuito).



En el esquema eléctrico anterior se ve más claro aún que este circuito no es más que la unión de dos circuitos independientes ya vistos anteriormente. Así pues, aparentemente, cada uno de estos dos circuitos no está relacionado con el otro, pero aquí es cuando interviene nuestro sketch, el cual funcionará como un "pegamento" entre las dos partes. Nuestro programa irá leyendo el estado del pulsador constantemente y cuando detecte que este esté siendo pulsado (al recibir una señal HIGH por el pin de entrada nº 8), reaccionará consecuentemente enviando una señal HIGH por el pin de salida nº 4 para encender el LED.

```
int estadoBoton=0; //Guardará el estado del botón (HIGH ó LOW)
void setup(){
	pinMode(4,OUTPUT); //Donde está conectado el LED
	pinMode(8,INPUT);  //Donde está conectado el pulsador
}
void loop() {
	estadoBoton=digitalRead(8);
//Si se detecta que el botón está pulsado, se enciende el LED
	if (estadoBoton == HIGH) {
		digitalWrite(4,HIGH);
//Si no, no
	} else {
		digitalWrite(4,LOW);
	}
}
```

A partir de aquí, debería ser bastante sencillo modificar el código anterior para que cada vez que, por ejemplo, se mantuviera pulsado el botón, el LED parpadeara cinco veces por segundo (es decir, estuviera 100 ms iluminado, 100 ms apagado, 100 ms iluminado, 100 ms apagado…y así) y cuando se soltara no se iluminara en absoluto. Se deja como ejercicio al lector.

Otra modificación del proyecto anterior (hay tantas como la imaginación nos permita) podría ser añadir al circuito un nuevo LED (junto con su correspondiente divisor de tensión) conectado al pin digital nº 5 (por ejemplo) y modificar el código anterior para que cuando se mantuviera pulsado el botón se encendiera un LED y cuando se soltara se encendiera el otro. El truco está en enviar una señal HIGH a un LED y una señal LOW al otro al mismo tiempo, según el estado detectado del pulsador. Se deja como ejercicio al lector.

Por otro lado, indicar que si hubiéramos querido realizar el ejemplo anterior usando la versión de pulsador con resistencia “pull-up”, deberíamos haber tenido en cuenta que si *digitalRead()* devuelve LOW, es cuando el botón está pulsado y cuando devuelve HIGH es cuando el botón no está activado. Nada más.

Ejemplo 6.9: Ahora lo que queremos es usar un botón que no se tenga que mantener pulsado para activar una salida, sino que pulsándolo una vez ya se active y pulsándolo otra vez se desactive. Pensemos por ejemplo en el botón de encendido/apagado de un mando a distancia del televisor: sería muy pesado tener que mantenerlo pulsado todo el rato para hacer funcionar el aparato.

Como ejemplo utilizaremos el mismo circuito de los códigos anteriores, con un pulsador conectado a una resistencia “pull-down” y a la entrada digital número 8 por un lado y con un LED conectado a un divisor de tensión y la salida digital número 4 por otro. El código, presentado a continuación, es lo que cambia:

```
int estadoActual=0; //Guarda el estado actual del botón
int estadoUltimo=0; //Guarda el último estado del botón
int contador=0; //Guarda las veces que se pulsa el botón
void setup(){
	pinMode(4,OUTPUT); //Donde está conectado el LED
	pinMode(8,INPUT);   //Donde está conectado el pulsador
	Serial.begin(9600);
}
void loop(){
	//Leo el nuevo estado actual del botón
	estadoActual=digitalRead(8);
	//Si éste cambia respecto el estado justo anterior…
	if (estadoActual != estadoUltimo){
/*…lo notifico. Ojo, tengo que comprobar que el cambio sea una
pulsación y no una liberación. Si hubiéramos usado una resistencia
pull-up, el valor a comprobar de “estadoActual” sería LOW */
		if (estadoActual == HIGH) {
			contador = contador + 1;
/*Notar que se ha de poner varios Serial.print para intercalar en una
sola sentencia frase literal con valores de variables*/
			Serial.print ("Ésta es la pulsación n° ");
			Serial.println(contador);
		}
	}
//Guardo el estado actual para la siguiente comprobación
	estadoUltimo= estadoActual;
```



```
/*Cambiamos el estado del LED contando las veces que se ha pulsado el
botón, de forma alternativa (número par = LED apagado, número impar =
LED encendido) ¡Ojo, no contamos cuando se suelta!)*/
      if (contador % 2 == 0 ) {
          digitalWrite(4, LOW);
      } else {
          digitalWrite(4, HIGH);
      }
}
```

Expliquemos el código anterior. Este sketch continuamente está monitorizando el estado del botón. Si detecta en algún momento que el botón sufre un cambio (es decir, si sufre una pulsación –pasa de no estar apretado a sí estarlo– o bien sufre una liberación –pasa de estar apretado a no estarlo–), se comprueba de cuál de estos dos cambios se trata (pulsación o liberación). Si es el segundo caso no hará nada, pero si se trata de una pulsación, se envía a través del canal serie un mensaje notificando que ha ocurrido dicha pulsación (hemos añadido esta nueva funcionalidad en nuestro sketch además de la del simple encendido de un LED), y sobre todo, se aumenta el contador de pulsaciones. Este contador será importante después, más allá de que ahora sea un simple numerito mostrado en el "Serial monitor". Antes de proceder con la manipulación de los LEDs, el sketch aún tiene que guardar el estado actual del botón (haya cambiado o no: por eso la línea correspondiente está fuera de cualquier "if") para que en la nueva repetición del "loop()" se compare este con el nuevo estado que tendrá a continuación, y así constantemente.

Para mantener el LED encendido (o apagado, según el caso) sin necesidad de mantener pulsado todo el rato el botón, se hace uso del contador de pulsaciones comentado antes. La clave está en darse cuenta de que solo nos interesan las pulsaciones y no las liberaciones (es decir, no todos los cambios del botón nos interesan: solo los que contamos con la variable "contador"). Lo que queremos es que (suponiendo que el LED está apagado al iniciar el sketch) cuando el botón se pulse una vez ("vez nº 1"), el LED se encienda, cuando se pulse la siguiente vez ("vez nº 2") se apague, cuando se pulse la siguiente vez ("vez nº 3") se encienda, y así. Podemos fijarnos que se cumple un patrón: si la pulsación ocurre una vez impar, el LED se encenderá y si la vez es par, se apagará. Por tanto, la idea es comprobar que el contador de pulsaciones tenga un valor par o impar. A medida que se realicen pulsaciones este contador irá aumentando hasta llegar a su límite máximo marcado por su tipo de datos, pero en ese momento su valor se reseteará por debajo y seguirá aumentando, así que en este sentido no habrá ningún problema.

¿Y cómo se sabe si un número es par o impar? Dividiéndolo entre dos y observando el resto de la división: si este es 0, el número es par. El lenguaje Arduino consta de un operador matemático (llamado "módulo" y representado con el signo %) que permite obtener precisamente el resto de una división. Así que ya lo tenemos todo.

Como ejercicio: ¿cómo se podría modificar el código anterior para que mostrara por el "Serial monitor" una cuenta atrás de 10 pulsaciones (por ejemplo) y que al llegar a 0 (es decir, al hacer diez pulsaciones) se imprimiera un mensaje final? El truco está en usar un valor inicial para la variable "contador" igual a 10 y modificar el interior de la sección *if (estadoActual == HIGH) {}* con la introducción allí de dos nuevos "ifs" : uno para comprobar si esa variable es aún mayor que 0 (en cuyo caso, se disminuiría el valor de "contador" y se mostraría la correspondiente cuenta atrás), y otro para comprobar si esa variable es exactamente igual que 0 (mostrando entonces el mensaje final). ¿Se seguiría encendiendo el LED una vez mostrado el mensaje final?

Volviendo al sketch de ejemplo, desgraciadamente, es posible que al probarlo se vea que cada pulsación del botón genera más de un mensaje por el "Serial monitor" y que el LED a veces no reacciona a las pulsaciones (de hecho, ambos problemas están relacionados). Esto ocurre porque durante el primer milisegundo de cada presionado (y soltado) del botón se producen a nivel electrónico pequeñas variaciones de la señal de entrada que hacen que los valores HIGH y LOW obtenidos del pulsador alternen rápidamente hasta que no se estabilizan en el valor adecuado.

Este fenómeno se llama "bounce" (rebote) y es inevitable por la propia construcción de este tipo de pulsadores: cuando el botón se aprieta, una lámina existente bajo este es presionada y hace contacto con dos extremos conductores; cuando el botón se deja de apretar, esta lámina retorna. Pero durante el primer milisegundo de la pulsación, esta lámina puede rebotar varias veces entre sus dos posiciones hasta que queda fijada finalmente en su posición correcta, provocando por tanto lecturas alternas del estado del pulsador. Esto causa que a veces nuestro sketch no recoja el valor correcto del estado del pulsador y parezca estar recibiendo múltiples pulsaciones y liberaciones ficticias, y por tanto, múltiples valores HIGH y LOW sin sentido.

Existen muchas soluciones para resolver este problema: incluir en el circuito un condensador (con un terminal conectado entre el pulsador y el pin de entrada de la placa y el otro conectado a tierra) para "amortiguar" la señal rebotada, usar otro tipo de interruptores más sofisticados, etc. Sin embargo, la solución más sencilla es realizar dos lecturas del estado del botón con una diferencia de unos milisegundos entre ellas. Si en las dos lecturas obtenemos valores diferentes, significa que estamos

en un momento de rebote y por tanto nuestro sketch no debe hacer nada hasta que vuelva a comprobar si la señal ya está estabilizada; si las dos lecturas son la misma, significa que el estado del botón finalmente es "el que ha de ser" (es decir, no lo hemos pillado por un rebote de casualidad) y nuestro sketch podrá tomar por tanto dicho valor como bueno.

Ejemplo 6.10: Para solucionar el fenómeno del "bounce", modificaremos el sketch del ejemplo anterior. Tal como se puede observar en el código mostrado a continuación, el truco está en realizar dos lecturas del estado del pulsador separadas por 10 milisegundos (tiempo suficiente). Entonces se comprueba que ambas lecturas sean iguales. Si es así, se deduce que el pulsador está en una posición estable y por tanto podemos ejecutar toda la lógica del conteo de pulsaciones y del encendido del LED (código que es idéntico al del ejemplo anterior). Notar que hemos añadido una nueva variable ("estadoActual2") que representa el valor obtenido en la segunda lectura y a la variable llamada "estadoActual" en el ejemplo anterior la hemos llamado ahora "estadoActual1".

```
int estadoActual1=0;
int estadoActual2=0;
int estadoUltimo=0;
int contador=0;
void setup(){
      pinMode(4,OUTPUT);
      pinMode(8,INPUT);
      Serial.begin(9600);
}
void loop(){
      estadoActual1=digitalRead(8);
      delay(10);
      estadoActual2=digitalRead(8);
//Si los estados no son iguales, el sketch no hace gran cosa
      if (estadoActual1 == estadoActual2) {
//El código que sigue es idéntico al del anterior ejemplo
          if (estadoActual1 != estadoUltimo){
              if (estadoActual1 == HIGH) {
                   contador = contador + 1;
                   Serial.print ("Ésta es la pulsación n° ");
                   Serial.println(contador);
              }
          }
      }
      estadoUltimo= estadoActual1;
```

```
      if (contador % 2 == 0 ) {
          digitalWrite(4, LOW);
      } else {
          digitalWrite(4, HIGH);
      }
}
```

<u>Ejemplo 6.11</u>: Vamos a diseñar ahora otro circuito: un juego. El circuito constará de tres LEDs conectados cada uno (a través de su respectivo divisor de tensión en serie) a un pin digital diferente de la placa Arduino. Estos LEDs se irán encendiendo y apagando de forma secuencial, y cuando el LED del medio se encienda, el jugador debe apretar en un pulsador. Si acierta, se mostrará un mensaje por el "Serial monitor" y la velocidad de la secuencia de iluminación de los LEDs aumentará (y también lo hará por tanto la dificultad). En nuestro sketch los LEDs están conectados a los pines digitales 5, 6 y 7, y el pulsador al pin 8. El tiempo inicial entre encendido y encendido de los LEDs es 200 ms, pero si el jugador acierta, este tiempo disminuirá en 20 ms, hasta llegar a un tiempo entre encendidos de 10 ms, momento en el cual se volverá al tiempo inicial de 200 ms.

```
int leds[]={5,6,7};
int i=0;
int tiempo=200;
void setup (){
      for(i=0;i<=2;i++) {
            pinMode(leds[i],OUTPUT);
      }
      pinMode(8,INPUT);
      Serial.begin(9600);
}
void loop () {
//Recorro los LEDs del array, iluminándolos y apagándolos
      for(i=0;i<=2;i++) {
            digitalWrite(leds[i],HIGH);
            delay(tiempo);
//Antes de apagar cada LED, miro si el jugador ha acertado
            compruebaacierto();  //Función propia
            digitalWrite(leds[i],LOW);
            delay(tiempo);
      }
}
void compruebaacierto(){
/*Si se tiene pulsado el botón y ahora mismo el LED encendido es de
índice 1 dentro del array (el del medio), se acertó */
```

```
        if(digitalRead(8)==HIGH && i==1) {
             Serial.println("Acierto");
               tiempo=tiempo-20;
               if(tiempo<10){
                      tiempo=200;
               }
        }
}
```

Ejemplo 6.12: Evidentemente, con pulsadores no solo podemos controlar LEDs, sino cualquier otro tipo de actuador. Por ejemplo, en un circuito con dos pulsadores conectados a los pines de entrada digital nº 7 y nº 8 respectivamente (además de a la alimentación y a tierra a través de una resistencia “pull-down”) y un servomotor conectado al pin de salida PWM nº 3 (además de a la alimentación y a tierra), podríamos ejecutar el siguiente código. Gracias a él, pulsando un botón el servomotor se movería en un sentido de giro, y pulsando el otro botón se movería en sentido contrario.

```
#include <servo.h>
Servo miservo;
int pos = 90;  //Posición del servo
void setup()  {
       pinMode(7, INPUT);
       pinMode(8, INPUT);
       miservo.attach(3);
       miservo.write(pos);//Posición inicial en el centro
}
void loop() {
       if(digitalRead(7) == HIGH)  {
             if( pos > 0)  {
                    pos--;
                 //Mueve el servo de 180 a 0 grados
                    miservo.write(pos);
              }
       if(digitalRead(8) == HIGH)  {
              if( pos < 180)  {
                    pos=pos++;
                 //Mueve el servo de 0 a 180 grados
                    miservo.write(pos);
              }
       }
}
```

Ejemplo 6.13: Otro circuito sencillo que podemos realizar con solo dos pulsadores (conectados a los pines de entrada digital nº 7 y nº 8 respectivamente, además de a la alimentación y a tierra a través de una resistencia "pull-down") es el de un temporizador, donde un botón servirá para poner en marcha la cuenta de tiempo y el otro para pararla. A través del canal serie se mostrará el número de horas, minutos y segundos transcurridos entre ambas pulsaciones.

```
unsigned long inicio, fin, transcurrido;
void setup(){
  Serial.begin(9600);
  pinMode(7, INPUT); //Botón de inicio
  pinMode(8, INPUT); //Botón de fin
}
void loop(){
  if (digitalRead(7)==HIGH){
    inicio=millis();
    delay(200); //Para hacer el "debounce"
  }
  if (digitalRead(8)==HIGH){
    fin=millis();
    delay(200); //Para hacer el "debounce"
    verResultado();
  }
}
void verResultado(){
  float h,m,s,ms;
  unsigned long resto;
  transcurrido=fin-inicio;
  h=int(transcurrido/3600000); //Número de horas
//Obtengo el resto de ms sobrantes que no llegan a una hora
  resto=transcurrido%3600000;
  m=int(resto/60000); //Número de minutos
//Obtengo el resto de ms que no llegan a un minuto
  resto=resto%60000;
  s=int(resto/1000);  //Número de segundos
//Obtengo el resto de ms que no llegan a un segundo
  ms=resto%1000;
  Serial.print("Total de milisegundos transcurridos");
  Serial.println(transcurrido);
  Serial.print("Tiempo transcurrido formateado");
  Serial.print(h);  Serial.print("h ");
  Serial.print(m);  Serial.print("m ");
  Serial.print(s);  Serial.print("s ");
  Serial.print(ms); Serial.println("ms");
```



```
  Serial.println();
}
```

Ejemplo 6.14: Otro circuito curioso es el siguiente: se trata de implementar el juego de "los trileros", en el cual tenemos tres LEDs que durante un breve lapso de tiempo se iluminan en una secuencia rápida y aleatoria. El usuario deberá adivinar cuál de los tres LEDs es el último en iluminarse apretando el pulsador correspondiente. Existe un pulsador por cada LED, y en el código se han configurado con las resistencias "pull-up" internas de la placa Arduino, por lo que su conexión no requiere ninguna resistencia externa, tal como ya se ha comentado en párrafos anteriores. Si el usuario acierta, se enviará un mensaje de felicitación por el canal serie; si no, se enviará un mensaje de consuelo.

```
byte LEDizq = 2;   //Pin-hembra donde está conectado el LED izquierdo
byte LEDmedio = 3; //Pin-hembra donde está conectado el LED del medio
byte LEDder = 4;   //Pin-hembra donde está conectado el LED derecho
byte BotonIzq = 7; //Pin-hembra donde está conectado el pulsador izq.
byte BotonMed = 8; //Pin-hembra donde está conectado el pulsador med.
byte BotonDer = 9; //Pin-hembra donde está conectado el pulsador der.
byte LEDrandom;
byte LEDultimo;
byte respuesta;
void setup() {
  pinMode(LEDizq, OUTPUT);
  pinMode(LEDmedio, OUTPUT);
  pinMode(LEDder, OUTPUT);
  pinMode(BotonIzq, INPUT_PULLUP);
  pinMode(BotonMedio, INPUT_PULLUP);
  pinMode(BotonDer, INPUT_PULLUP);
  Serial.begin(9600);
}
void loop() {
//Uso como semilla aleatoria el ruido leído en una entrada analógica
  randomSeed(analogRead(0));
  //Enciendo los tres LEDs de forma aleatoria
  for(int x = 0; x < 10; x++) {
    LEDrandom = random(2,5);
    digitalWrite(LEDrandom, HIGH); delay(15);
    digitalWrite(LEDrandom, LOW);  delay(10);
    if(x == 9) {
      LEDultimo = LEDrandom;
    }
  }
/*Me espero mientras no se reciba ninguna respuesta. La condición del
```

```
while siempre es cierta (se trata de un bucle infinito), pero cuando
se detecte una pulsación, se sale de él y se continúa el programa */
  while(1 == 1) {
    if(digitalRead(BotonIzq) == LOW) {
      respuesta = 2; break;
    } else if(digitalRead(BotonMedio) == LOW) {
      respuesta = 3; break;
    } else if(digitalRead(BotonDer) == LOW) {
      respuesta = 4; break;
    }
  }
  //Compruebo si la respuesta es correcta o no
  if(respuesta == LEDultimo) {
      Serial.print("Muy bien");
  } else {
      Serial.print("Vaya, lo siento");
  }
}
```

Keypads

Un paso más allá en el uso de pulsadores digitales son los teclados numéricos como el de la imagen inferior (también llamados "keypads"). En efecto, estos aparatos no son más que pulsadores conectados entre sí de tal forma que, cuando alguno de ellos es apretado, envía una señal identificativa de tipo binario a nuestra placa Arduino para que así esta pueda reaccionar de la forma que nosotros deseemos dependiendo del botón pulsado. Una aplicación práctica podría ser la introducción de una secuencia determinada de dígitos (una "contraseña") para compararla con una ya previamente definida en el código y así activar (o no) algún componente del circuito.



Si observamos el dorso de un keypad típico, veremos que ofrece una serie de pines numerados (entre siete y nueve normalmente) a los que se les ha de conectar los pines-hembra de la placa Arduino (bien directamente o bien a través de una breadboard o PCB). Si el keypad es de 3x4 (como el de la imagen; uno similar es el producto nº 8653 de Sparkfun) suele ofrecer como mínimo siete pines, donde cada uno de ellos se corresponde o bien a una fila de las cuatro o una columna de las tres. También existen keypads de 4x4 y otras dimensiones. En todo caso, el sistema siempre es el mismo: cuando se pulse un

botón determinado del keypad, el pin correspondiente a su fila y el correspondiente a su columna enviarán una señal digital HIGH a nuestra placa Arduino, identificando inequívocamente a la tecla pulsada, ya que solo existe una pareja posible de valores (fila, columna) para cada botón. Para saber qué pines se corresponden con qué fila o columna, se ha de consultar su datasheet.

Si no disponemos del datasheet de nuestro keypad, aún podemos encontrar la relación de los pines con las filas/columnas utilizando la funcionalidad de medir continuidad con un multímetro. Se trataría de conectar un cable del multímetro en el pin nº 1 del keypad y otro en el pin nº 2, e ir pulsando los diferentes botones, hasta escuchar (si se produce) el zumbido que indicara que hay continuidad; entonces pasaríamos a conectar el segundo cable del multímetro al siguiente pin (el nº 3 en este caso) y repetiríamos el proceso. Después conectaríamos ese mismo segundo cable al nº 4, y así hasta el final. Entonces pasaríamos a conectar el primer cable al pin nº 2 y el segundo cable al 3,4,5,6…; luego el primer cable al nº 3 y el segundo cable al 4,5,6... y así con todas las parejas de pines posibles, anotando siempre qué botón provoca continuidad en una combinación dada. Es posible que en este procedimiento se descubra que hayan pines que no "sirvan para nada"; se pueden obviar tranquilamente.

En el caso concreto del keypad de Sparkfun, tenemos que conectar cada uno de sus siete pines a un pin-hembra digital de la placa Arduino diferente (los cuales actuarán como entradas) y además, para alimentar el keypad, debemos conectar sus pines nº 3, 5, 6 y 7 al pin "5V" de la placa Arduino a través de un divisor de tensión de 1 KΩ ó 10 KΩ.

Para facilitar el uso de este tipo de dispositivos, lo más recomendable es utilizar la librería "Keypad", librería no oficial, pero disponible dentro de la propia web de Arduino (concretamente, en http://arduino.cc/playground/Code/Keypad). Es una librería no bloqueante (es decir: las pulsaciones realizadas en el keypad no interrumpen el funcionamiento normal del microcontrolador) y permite pulsaciones de múltiples teclas a la vez. Una vez instalada como cualquier otra librería, y conectado cada pin del keypad a un pin-hembra digital de la placa Arduino configurado como entrada, ya podemos empezar a usarla. Desgraciadamente, no hay espacio suficiente en el libro para mostrar su funcionamiento, así que remito a su documentación oficial (descargada junto con los ficheros que componen la librería y complementada con varios ejemplos de código).

USO DE LAS ENTRADAS Y SALIDAS ANALÓGICAS

Las funciones que sirven para gestionar entradas y salidas analógicas son las siguientes:

analogWrite(): envía un valor de tipo "byte" (especificado como segundo parámetro) que representa una señal PWM, a un pin digital configurado como OUTPUT (especificado como primer parámetro). No todos los pines digitales pueden generar señales PWM: en la placa Arduino UNO por ejemplo solo son los pines 3, 5, 6, 9, 10 y 11 (están marcados en la placa). Cada vez que se ejecute esta función se regenerará la señal. Esta función no tiene valor de retorno.

Recordemos que una señal PWM es una señal digital cuadrada que simula ser una señal analógica. El valor simulado de la señal analógica dependerá de la duración que tenga el pulso digital (es decir, el valor HIGH de la señal PWM). Si el segundo parámetro de esta función vale 0, significa que su pulso no dura nada (es decir, no hay señal) y por tanto su valor analógico "simulado" será el mínimo (0V). Si vale 255 (que es el máximo valor posible, ya que las salidas PWM tienen una resolución de 8 bits, y por tanto, solo pueden ofrecer hasta 2^8=256 valores diferentes –de 0 a 255, pues–), significa que su pulso dura todo el período de la señal (es decir, es una señal continua) y por tanto su valor analógico "simulado" será el máximo ofrecido por la placa (5 V). Cualquier otro valor entre estos dos extremos (0 y 255) implica un pulso de una longitud intermedia (por ejemplo, el valor 128 generará una onda cuadrada cuyo pulso es de la misma longitud que la de su estado bajo) y por tanto, un valor analógico "simulado" intermedio (en el caso anterior, 2,5 V).

Es importante recalcar que esta función no tiene nada que ver con los pines analógicos A0, A1, etc., ya que estos solo funcionan como pines analógicos de entrada (mediante el uso de la función *analogRead()*) pero no de salida. Las salidas analógicas se han de generar utilizando solamente los pines digitales PWM.

analogRead(): devuelve el valor leído del pin de entrada analógico cuyo número (0, 1, 2...) se ha especificado como parámetro. Este valor se obtiene mapeando proporcionalmente la entrada analógica obtenida (que debe oscilar entre 0 y un voltaje llamado voltaje de referencia, el cual por defecto es 5 V) a un valor entero entre 0 y 1023. Esto implica que la resolución de lectura es de 5V/1024, es decir, de 0,049 V.

Ya comentamos en el capítulo 2 que es posible aumentar la resolución de lectura (es decir, detectar cambios de voltaje de entrada más pequeños) si se reduce

el voltaje de referencia. Esto se hace mediante la función *analogReference()*, explicada en los próximos párrafos. También comentamos en el capítulo 2 que este proceso de mapeado lo realiza un convertidor analógico-digital interno incorporado a la placa, que tiene 6 canales (por eso hay 6 pines analógicos) y que tiene 10 bits de resolución (por eso los valores finales van de 0 a 1023: 2^{10} valores posibles).

Como los pines analógicos por defecto solamente funcionan como entradas de señales analógicas, no es necesario utilizar previamente la función *pinMode()* con ellos. No obstante, estos pines también incorporan toda la funcionalidad de un pin de entrada/salida digital estándar (incluyendo las resistencias "pull-up"), por lo que si se necesita utilizar más pines de entrada/salida digitales de los que la placa Arduino ofrece, y los pines analógicos no están en uso, estos pueden ser utilizados como pines de entrada/salida digitales extra de la forma habitual, simplemente identificándolos con un número correlativo más allá del pin 13, que es el último pin digital. Es decir, el pin "A0" sería el número 14, el "A1" sería el 15, etc. Por ejemplo, si quisiéramos que el pin analógico "A3" funcionara como salida digital y además enviara un valor BAJO, escribiríamos primero `pinMode(17,OUTPUT);` y luego `digitalWrite(17,LOW);`

Si un pin analógico no está conectado a nada, el valor devuelto por *analogRead()* fluctuará debido a múltiples factores como por ejemplo los valores que puedan tener las otras entradas analógicas, o lo cerca que esté nuestro cuerpo a la placa, etc. Esto, que en principio no es deseable, lo podemos utilizar sin embargo para algo útil: para establecer semillas de números aleatorios diferentes (y por tanto, aumentar así la aleatoriedad de las diferentes series de números generados). Esto se haría poniendo como parámetro de *randomSeed();* el valor obtenido en la lectura de un pin analógico cualquiera que esté libre; es decir, por ejemplo así: `randomSeed(analogRead(0));`

Por otro lado, hay que saber que el convertidor analógico/digital tarda alrededor de 100 microsegundos (0,0001s) en procesar la conversión y obtener el valor digital, por lo que el ritmo máximo de lectura en los pines analógicos es de 10000 veces por segundo. Esto hay que tenerlo en cuenta en nuestros sketches.

Solo si disponemos de la placa Arduino Due, podremos hacer servir otras dos funciones más relacionadas con las entradas y salidas digitales:

analogWriteResolution(): establece, mediante su único parámetro –de tipo "byte"–, la resolución en bits que tendrá a partir de entonces la función *analogWrite()* a lo largo de nuestro sketch. Este parámetro puede ser un número entre 1 y 32. Por defecto, esta resolución es de 8 bits (es decir, que

con *analogWrite()* se pueden escribir valores entre 0 y 255), pero la placa Arduino Due dispone de dos conversores digital-analógicos que permiten trabajar con una resolución de hasta 12 bits. Esto significa que, si usamos estas dos salidas analógicas especiales y usamos *analogWriteResolution()* para establecer a 12 la resolución deseada, la función *analogWrite()* podría llegar a escribir valores de entre 0 y 4095. Esta función no devuelve nada.

Si en *analogWriteResolution()* se especifica una resolución mayor de la que las salidas analógicas de la placa son capaces de admitir, los bits extra serán descartados. Si, en cambio, se especifica una resolución menor, los bits extra se rellenarán con ceros. Por ejemplo: si escribimos `analogWriteResolution(16);`, solo los primeros 12 bits de cada valor (empezando por la derecha) serán utilizados por *analogWrite()*, y los últimos 4 no se tendrán en cuenta. Si, en cambio, escribimos `analogWriteResolution(8);`, se añadirán automáticamente 4 bits (iguales a 0) a la izquierda del valor de 8 bits, para que así *analogWrite()* pueda escribir, a través de los dos conversores digital-analógicos, un valor de 12 bits.

analogReadResolution(): establece, mediante su único parámetro –de tipo "byte"–, el tamaño en bits del valor que devolverá la función *analogRead()* a partir de entonces a lo largo de nuestro sketch. Este parámetro puede ser un número entre 1 y 32. Por defecto, este tamaño es de 10 bits (es decir, que *analogRead()* devuelve valores entre 0 y 1023). La placa Arduino Due es capaz de manejar tamaños de hasta 12 bits en los valores devueltos por *analogRead()*, por lo que podríamos obtener datos dentro de un rango de entre 0 y 4095. Esta función no devuelve nada.

Si en *analogReadResolution()* se especifica un tamaño mayor del que las entradas analógicas de la placa son capaces de devolver, al valor leído se le añadirán por la izquierda bits extra iguales a 0 hasta llegar al tamaño especificado. Si, en cambio, se especifica una resolución menor, los bits leídos sobrantes (por la izquierda) no se tendrán en cuenta y serán descartados.

Ejemplos con salidas analógicas

Hasta ahora hemos conectado siempre los LEDs a salidas digitales. Por tanto, estos solo podían estar en dos estados: o apagados o encendidos. Pero los LEDs (como tantos otros) son dispositivos analógicos. Esto implica que pueden tener muchos más estados (de hecho, pueden tener estados continuos). Es decir, que pueden iluminarse con muchas intensidades diferentes y de forma gradual.

Ejemplo 6.15: Para conseguir cambiar la intensidad lumínica de un LED, primero deberemos montar el circuito. La conexión del LED no tiene ningún misterio: su terminal positivo ha de ir enchufado a un pin PWM de nuestra placa Arduino (por ejemplo el nº 9) y su terminal negativo a tierra. Se recomienda también conectarle en serie un divisor de tensión (de unos 220 ohmios está bien). A continuación, debemos escribir el sketch, el cual modificará el voltaje PWM ofrecido por la salida analógica donde esté conectado el LED (recordemos, de 0 –valor mínimo– a 255 –valor máximo–).

```
int brillo = 0;
int incremento = 5; //Puede valer negativo (decremento)
void setup(){
    pinMode(9, OUTPUT); //El LED está conectado al pin 9 (PWM)
}
void loop()  {
    analogWrite(9, brillo);
    /*Cambio el brillo un incremento dado
    (se verá en el próximo analogWrite) */
    brillo = brillo + incremento;
    /*Si se llega a los extremos, se invierte
    la dirección del incremento */
    if (brillo == 0 || brillo == 255) {
          incremento = -incremento;
    }
/*Espero 30 milisegundos con la señal actual de analogWrite() para
ver mejor el efecto. Si se desea que el (incremento o disminución)
del brillo se realice a otra velocidad, basta con modificar el tiempo
de espera */
    delay(30);
}
```

Otro código que hace exactamente lo mismo es:

```
int brillo = 0;
void setup(){
pinMode(9, OUTPUT); //El LED está conectado al pin 9 (PWM)
}
void loop(){
    //Incrementa el brillo (de mínimo a máximo)
    for(brillo = 0 ; brillo<= 255; brillo=brillo+5) {
         analogWrite(9, brillo);
         delay(30);
    }
```

```
    //Disminuye el brillo (de máximo a mínimo)
    for(brillo = 255; brillo>=0; brillo=brillo-5) {
         analogWrite(9, brillo);
         delay(30);
    }
}
```

Ejemplo 6.16: Podríamos incluso manipular el brillo de un LED a voluntad mediante el envío de algún comando adecuado a través del canal serie. En el siguiente ejemplo supondremos que tenemos 3 LEDs diferentes (de colores primarios rojo, verde y azul, respectivamente) conectados cada uno a través de su divisor de tensión correspondiente a un pin PWM diferente. La idea es utilizar el "Serial monitor" para enviar a la placa Arduino un comando determinado que especifique qué LED en concreto queremos manipular (indicado por la letra "r", "g" o "b") y qué cantidad de brillo le queremos asignar (indicado por un valor entre 0 y 255). Así, si queremos por ejemplo apagar el LED rojo el comando a enviar debería ser "r0". Y si queremos iluminar al máximo el LED azul, el comando debería ser "b255".

```
char color; //Es importante que sea de tipo char y no byte
byte brillo;
byte LedRojo  = 5;
byte LedVerde = 6;
byte LedAzul  = 9;
void setup() {
  Serial.begin(9600);
  pinMode(LedRojo,  OUTPUT);
  pinMode(LedVerde, OUTPUT);
  pinMode(LedAzul,  OUTPUT);
  analogWrite(LedRojo,  127);  //Establezco un brillo inicial medio
  analogWrite(LedVerde, 127);
  analogWrite(LedAzul,  127);
}
void loop () {
  //Leo el primer carácter recibido por el canal serie
  if (Serial.available()>0) {
    color=Serial.read();
    if( color == 'r' || color == 'g' || color == 'b' ) {
        delay(5);
        //Extraigo el número que sigue a la primera letra
        brillo=byte(Serial.parseInt());
        if(color == 'r') {
            analogWrite(LedRojo, brillo);
```



```
        } else if(colorCode == 'g'){
            analogWrite(LedVerde, brillo);
        }else if(colorCode == 'b'){
            analogWrite(LedAzul, brillo);
        }
    }
  }
  delay(100);  //Me espero para recibir los datos serie bien
}
```

Ejemplo 6.17: Otro código que persigue el mismo objetivo que el anterior, pero de una forma algo más compacta, es el siguiente. En él también enviamos a través del canal serie la cantidad de brillo que deseamos que tenga cada LED, pero lo hacemos enviando siempre el brillo de los tres LEDS en cada “comando”. La idea es enviar los tres brillos separados por comas o cualquier otro carácter (o cadena, incluso), de tal forma que Arduino recoja uno tras otro el primer brillo, el segundo y el tercero, para seguidamente asignarlos a los LEDs adecuados.

```
byte brilloRojo, brilloVerde, brilloAzul;
byte LedRojo  = 5;
byte LedVerde = 6;
byte LedAzul  = 9;
void setup() {
  Serial.begin(9600);
  pinMode(LedRojo,  OUTPUT);
  pinMode(LedVerde, OUTPUT);
  pinMode(LedAzul,  OUTPUT);
  analogWrite(LedRojo,  127);
  analogWrite(LedVerde, 127);
  analogWrite(LedAzul,  127);
}
void loop () {
  //Leo todos los caracteres recibidos por el canal serie
  while (Serial.available()>0) {
    //Busco el siguiente número entero válido en los datos entrantes
    brilloRojo = Serial.parseInt();
    //Repito la búsqueda
    brilloVerde = Serial.parseInt();
    //Repito la búsqueda
    brilloAzul = Serial.parseInt();
    //Miro si ha llegado el carácter de nueva línea: eso marca el fin
    if (Serial.read() == '\n') {
      //Restrinjo los valores leídos al rango 0-255, por si acaso
```

```
      brilloRojo = constrain(brilloRojo, 0, 255);
      brilloVerde = constrain(brilloVerde, 0, 255);
      brilloAzul = constrain(brilloAzul, 0, 255);
      //Ilumino los LEDs convenientemente
      analogWrite(LedRojo, brilloRojo);
      analogWrite(LedVerde, brilloVerde);
      analogWrite(LedAzul, brilloAzul);
      }
  }
}
```

Podemos jugar a establecer los valores PWM de los tres LEDs anteriores (rojo, verde y azul) de forma que se puedan obtener otros colores. Para ello es necesario colocar los tres LEDs físicamente muy próximos y es recomendable, para conseguir un mejor efecto, difuminar la luz a través de algún material tal como pañuelos de papel, por ejemplo. Si variamos entonces el brillo de uno de los LEDs (o de dos o de los tres a la vez) alteraremos la combinación total y obtendremos, a partir de los tres colores primarios, una mezcla que producirá un color totalmente diferente (amarillo, lila, naranja...). Incluso se pueden generar transiciones de un color a otro si vamos modificando el brillo de cada LED (mediante bucles “for”, por ejemplo).

Para generar colores no primarios, también se puede utilizar un LED RGB, que no es más que un LED con cuatro terminales. Cuidado porque pueden ser de dos tipos: si es de tipo “cátodo común” (como el producto nº 9264 de Sparkfun), el terminal más largo se conecta a tierra, y los tres restantes se conectan (a través de sendos divisores de tensión) a diferentes pines PWM de nuestra placa Arduino: uno servirá para recibir (usando un divisor de 220 Ω por ejemplo) la intensidad deseada de color rojo, otro para recibir (usando otro divisor de 100 Ω por ejemplo) la intensidad de color verde y otro para recibir (usando otro divisor de 100 Ω) la intensidad de color azul. Si es de tipo “ánodo común” (como el producto nº 159 de Adafruit), el terminal más largo se conecta a la alimentación (5V) y los tres restantes se conectan (igualmente a través de un divisor de tensión en serie) a tierra y además a diferentes pines PWM de nuestra placa Arduino. Hay que tener en cuenta además que en los LEDs de ánodo común, el valor enviado mediante *analogWrite()* está invertido: un valor 0 hace brillar al máximo el color correspondiente y un valor 255 lo apaga.

Ejemplo 6.18: Un código que muestra el uso de un LED RGB es el siguiente:

```
int pinRojo = 6;
int pinVerde = 5;
int pinAzul = 3;
```



```
void setup () {
  pinMode(pinRojo, OUTPUT);
  pinMode(pinVerde, OUTPUT);
  pinMode(pinAzul, OUTPUT);
}
void loop () {
  setColor(0, 0, 0);        // Apagado
  setColor(255, 0, 0);      // Rojo
  setColor(0, 255, 0);      // Verde
  setColor(0, 0, 255);      // Azul
  setColor(0, 255, 255);    // Cian
  setColor(255, 255, 0);    // Amarillo
  setColor(255, 0, 255);    // Fucsia
  setColor(255, 255, 255); // Blanco
}
void setColor (int rojo, int verde, int azul) {
  //Cátodo común
  //analogWrite(pinRojo, rojo);
  //analogWrite(pinVerde, verde);
  //analogWrite(pinAzul, azul);
  //Ánodo común
  analogWrite(pinRojo, 255 - rojo);
  analogWrite(pinVerde, 255 - verde);
  analogWrite(pinAzul, 255 - azul);
  delay(1000);
}
```

Una propuesta práctica de lo que acabamos de descubrir es la siguiente: cuando estudiemos los sensores de temperatura en el próximo capítulo, en vez de mostrar la temperatura a través del “Serial monitor” o algún otro método similar poco original, podríamos tener un sistema de LEDs mostrando un color rojo más intenso a más calor o un color azul más intenso a más frío.

Otra idea menos útil pero muy llamativa es la de enviar valores PWM aleatorios a uno o varios LEDs independientes (usando *random();* como segundo parámetro de *analogWrite();*) y entre envío y envío esperar también un tiempo aleatorio (usando *random();* como parámetro de *delay();*). Realmente el efecto es psicodélico.

Ejemplo 6.19: Para finalizar este apartado, presentamos un código donde interviene, además de un LED conectado al pin PWM nº 9, un pulsador en configuración “pull-down” cuya cable de control está conectado al pin digital nº 8. La novedad está en el

bucle *while*: lo que conseguimos con él es que mientras tengamos presionado el pulsador y no hayamos llegado al máximo de brillo, el LED se irá iluminando al ritmo que marque *delay()* debido al incremento del brillo de 5 puntos en cada vuelta. Una vez dejemos de pulsar el botón, de forma brusca apagaremos el LED.

```
int brillo=0;
void setup() {
	pinMode(9, OUTPUT);
	pinMode(8, INPUT);
}
void loop() {
	while (digitalRead(8) == HIGH && brillo<=255){
		analogWrite(9,brillo);
		delay(200);
		brillo=brillo+5;
	}
	brillo=0;
	analogWrite(9, 0);
}
```

¿Qué pasaría si en el código anterior sustituimos la palabra “while” por la palabra “if”? Que el LED nunca se encenderá, porque aunque la condición se cumpla y se haga el primer *analogWrite()* del interior del “if”, justo después de este se vuelve a resetear la variable “brillo” a cero, y por si fuera poco, el último *analogWrite()* apaga lo poco que se había podido iluminar, por lo que en cada repetición del “loop” volveremos a estar en las mismas.

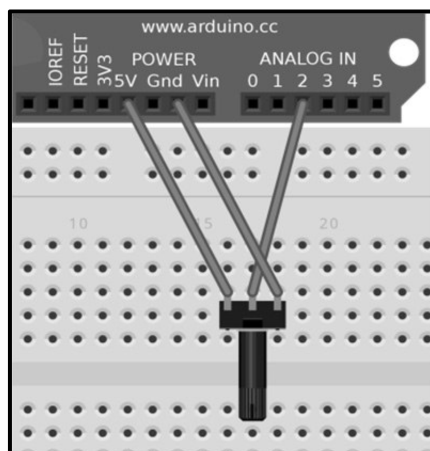
Ejemplos con entradas analógicas (potenciómetros)

Los ejemplos del apartado anterior están muy bien, pero nos gustaría modificar el brillo del LED a voluntad. Para ello, en nuestros circuitos podríamos utilizar un potenciómetro conectado por un extremo a una fuente de voltaje conocido (los 5 V ofrecidos por la propia placa Arduino ya nos va bien), por otro a tierra y por su patilla central a algún pin de entrada analógico de la placa. La idea es que la placa reciba (proveniente de esa patilla central) una señal analógica controlable a voluntad, y aprovechar esto para “reenviarla” a las salidas analógicas de la placa que queramos. En otras palabras: usaremos un potenciómetro conectado a una entrada analógica como intermediario para manipular un dispositivo conectado a alguna salida PWM (como un LED). Si no decimos lo contrario, en nuestros proyectos utilizaremos un potenciómetro de 10 KΩ.

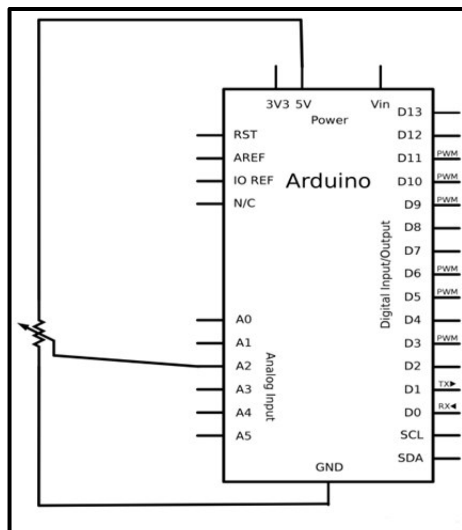
Ya sabíamos que un potenciómetro es una resistencia variable controlable por su patilla central, pero ¿cuál es esta señal analógica que recibe la placa Arduino proveniente de él? Esta señal es el voltaje existente entre la patilla central y el extremo del potenciómetro conectado a tierra. Por pura y simple Ley de Ohm, al variar la resistencia existente entre la patilla central y sus extremos, varía también la caída de potencial entre estos puntos. Concretamente, cuando entre la patilla central y el extremo del potenciómetro conectado a la alimentación haya una resistencia cercana a cero (y por tanto la resistencia entre la patilla central y el otro extremo, el conectado a tierra, sea máxima), el voltaje leído por la entrada analógica será cercano a 5 V. Cuando la patilla central esté en el otro lado, tocando al extremo conectado a tierra, la lectura será cercana a 0 V. Esta lectura del voltaje (analógico) controlable es el que hemos dicho que podremos "reenviar" a través de los pines PWM de la placa a los dispositivos que deseemos controlar analógicamente.

Recordemos por otro lado que la placa Arduino dispone de un conversor analógico-digital que solo permite utilizar valores entre 0 y 1023, por lo que el valor máximo del voltaje leído a través del potenciómetro (es decir, los 5 V) es convertido siempre al valor numérico 1023 (y el mínimo, lógicamente a 0). Los valores intermedios son convertidos proporcionalmente según la cantidad de voltaje recibido por el pin.

Ejemplo 6.20: Empezaremos por un circuito muy sencillo para ver en la práctica todo lo que se acaba de explicar. Consta tan solo de un potenciómetro y nada más, conectado tal como se ha dicho. El pin de entrada analógico escogido ha sido el nº 2.



El esquema eléctrico es realmente simple:



Si ejecutamos el siguiente sketch y movemos el potenciómetro, veremos por el “Serial monitor” las lecturas realizadas a través del pin analógico 2, que no son más que valores entre 0 y 1023. Notar que no se ha utilizado la función *pinMode()* como hasta ahora porque los pines-hembra analógicos de la placa Arduino solo pueden ser de entrada.

```
int valorPot=0;
void setup(){
     Serial.begin(9600);
}
void loop(){
     valorPot=analogRead(2);
     Serial.println(valorPot);
     delay(100);
}
```

De todas formas, tendría más gracia observar el valor analógico correspondiente a esa lectura. Es decir, ya sabemos que si vemos un 1023 este valor se corresponde con 5 V (esto es solo porque suponemos que estamos alimentando el potenciómetro con 5 V –los ofrecidos por la propia placa–), pero ¿y si vemos un 584? ¿Cuántos voltios se reciben en ese caso por la entrada analógica? Para saberlo, simplemente debemos aplicar una regla de proporcionalidad: multiplicar el valor leído por 5/1023.


```
int valorPot=0;
float voltajePot=0;
void setup(){ Serial.begin(9600);  }
void loop(){
     valorPot=analogRead(2);
/*La siguiente línea convierte el valor de valorPot en un valor de
voltaje. Fijarse que los valores obtenidos de analogRead() van desde
0 a 1023 y los valores que queremos van desde 0 a 5. Podríamos pensar
en utilizar la función map(), pero esta solo devuelve valores
enteros, por lo que no nos sirve. Afortunadamente, como en este caso
particular los mínimos de ambos rangos son 0, en vez de map() podemos
escribir una simple regla de proporcionalidad. Hay que tener en
cuenta el detalle de haber especificado los valores numéricos de la
fórmula como de tipo "float" (añadiéndoles el 0 decimal) para que el
resultado obtenido sea de tipo "float" también y no se trunque.*/
     voltajePot=valorPot*(5.0/1023.0);
     Serial.println(voltajePot);
     delay(100);
}
```

Con un poco de imaginación, podríamos modificar el ejemplo anterior para que en vez de mostrar el voltaje leído por el canal serie, lo visualizáramos en una tira de LEDs (10 por ejemplo), a modo de "termómetro" luminoso. Se deja como ejercicio.

Por otro lado, cuando trabajamos con sensores analógicos uno de los problemas que podemos tener es que cada cierto tiempo obtengamos algún "pico", es decir, un valor distorsionado y separado de los demás , y por tanto, erróneo. Es decir, ruido. Para intentar "suavizar" los valores leídos por si aparece alguno demasiado errático, podemos hacer servir un truco: leer la entrada analógica repetidas veces y calcular la media de los valores leídos, considerando esta el valor medido fiable.

Ejemplo 6.21: Para probar esta manera de obtener datos podemos utilizar el mismo circuito del ejemplo anterior. Concretamente, el código siguiente guarda diez lecturas analógicas en un array de diez posiciones, una a una. Por cada nuevo valor guardado, suma todos los valores y divide el resultado por el número de elementos del array (es decir, calcula la media de esos valores en ese preciso momento). Esta media, mostrada en el "Serial monitor", ofrece una lectura más suavizada del conjunto de valores leídos. Como la media se calcula cada vez que se lee un nuevo valor (en vez de esperar a llenar el array de diez valores nuevos, que sería otra manera), no se aprecia ningún tiempo de espera en los cálculos. Lógicamente, cuanto mayor sea el

número de elementos del array, mayor suavizado habrá en el resultado final, pero también será más lenta la obtención de este.

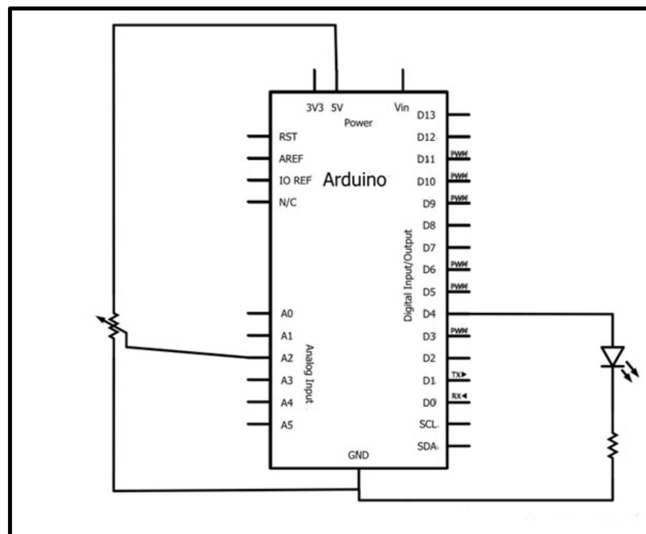
```
const int numElementos = 10; //Número de elementos del array
int lecturas[numElementos];  //Aquí se guardan las lecturas
int index = 0;   //Índice para irse moviendo por los elementos
 /*Valor de la suma de los 10 valores
 que haya en un momento dado en el array*/
int total = 0;
int media = 0;  // Es igual a total/numElementos
void setup(){
      Serial.begin(9600);
      //Inicializo todos los elementos del array a 0
      for (i = 0; i < numElementos; i++){
          lecturas[i] = 0;
      }
}
void loop() {
/*Quito de la suma total el valor que será sobrescrito enseguida por
el nuevo valor obtenido. De esta forma, se mantiene tan solo la suma
de los valores que en este momento estén dentro del array */
     total= total - lecturas[index];
     lecturas[index] = analogRead(2);
     //Añado el valor recién leído a la suma total
     total= total + lecturas[index];
/*Avanzo a la siguiente posición del array para
 sobrescribir en esta nueva posición el próximo valor leído*/
     index = index + 1;
     //Si estamos en el final del array…
     if (index >= numElementos){
           //…vuelvo al principio para sobrescribir por allí
          index = 0;
     }
     //Calculo la media de los 10 valores actuales
     media = total / numElementos;
     Serial.println(media);
     delay(1);//Me espero para leer de nuevo (por estabilidad}
}
```

Bien, volvamos al ejemplo 6.20: ya sabemos obtener un dato (de hasta 1024 valores posibles diferentes) proporcional al voltaje analógico recibido. La gracia está ahora en utilizar este valor de tensión para aplicársela a los dispositivos conectados a las salidas PWM de la placa Arduino. De esta manera, podremos variar gradualmente

el brillo de un LED, la velocidad de un motor, la frecuencia de un sonido emitido por un zumbador, etc., según les enviemos más o menos voltaje.

Por suerte, tanto el rango de voltaje analógico leído de entrada es de 5 V (porque hemos alimentado el potenciómetro con la propia placa) como también lo es el rango del voltaje de salida ofrecidos por los pines-hembra de Arduino, así que en este sentido no hay que preocuparse de que los valores leídos por un lado no sean eléctricamente seguros para ser utilizados en el otro. No obstante, hay que tener en cuenta que los valores leídos pueden estar entre 0 y 1024 (como ya hemos dicho) pero los valores PWM recordemos que solo pueden estar entre 0 y 255. Esto es muy importante, porque nos obliga en nuestro sketch a hacer un "mapeo" (normalmente con *map()*) de los valores obtenidos para que se ajusten a este nuevo rango cuatro veces más pequeño que el original.

Ejemplo 6.22: Todo esto lo probaremos en el siguiente circuito, donde añadiremos un LED (conectado a un pin de la placa que ha de ser de tipo PWM, como por ejemplo el nº 9) al potenciómetro que ya teníamos montado.



Y aquí está el código:

```
int valorPot=0;
void setup(){
      pinMode(9,OUTPUT);
}
```

```
void loop(){
     valorPot=analogRead(2);
/*Los valores de analogRead() van desde 0 a 1023 y los valores de
analogWrite van desde 0 a 255, por eso reajustamos el valor leído
para poderlo reenviar. En este caso particular, como los mínimos de
ambos rangos son 0, en vez de map() podríamos haber escrito una
simple regla de proporcionalidad, mediante la fórmula
valorPot=valorPot*(255.0/1023.0), o dicho de otra forma: dividiendo
el valor original entre 4.0 (el 4 ha de ser un valor “float” para que
el resultado no se trunque!, de ahí el 0 decimal).*/
     valorPot=map(valorPot,0,1023,0,255);
     analogWrite(9,valorPot);
//Espero un rato para que la señal de analogWrite se mantenga
     delay(100);
}
```

Ejemplo 6.23: En vez de utilizar la lectura del potenciómetro para variar de forma continua el brillo de un LED, otra cosa que podemos hacer con el mismo circuito del ejemplo anterior es enviar al LED una señal digital con *digitalWrite()* –es decir, sin valores intermedios: o se enciende (HIGH) o se apaga (LOW)– para hacerlo parpadear y utilizar entonces la lectura del potenciómetro como parámetro de *delay()* para establecer el tiempo de parpadeo. De esta forma, al variar de forma continua el estado del potenciómetro, variaremos de forma continua el tiempo de parpadeo:

```
int valorPot = 0;
void setup() {
     pinMode(9, OUTPUT);
}
void loop() {
     valorPot = analogRead(2);
     digitalWrite(9, HIGH);
     delay(valorPot);
     digitalWrite(9, LOW);
     delay(valorPot);
}
```

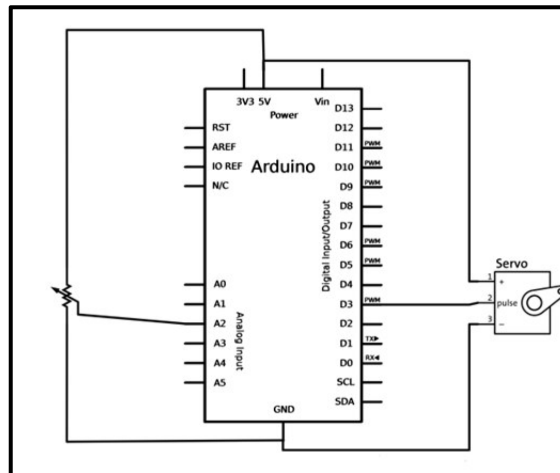
Ejemplo 6.24: O también encender el LED solamente si el valor leído del potenciómetro supera un determinado umbral:

```
int valorPot = 0;
void setup() {
     pinMode(9, OUTPUT);
```



```
}
void loop() {
     valorPot = analogRead(2);
     if (valorPot > 500) {
         digitalWrite(9, HIGH);
     } else {
         digitalWrite(9, LOW);
     }
     delay(valorPot);
}
```

Ejemplo 6.25: Evidentemente, además de LEDs, mediante un potenciómetro podemos controlar cualquier otro tipo de actuador, como por ejemplo un servomotor. Si diseñamos un circuito tal como el mostrado en la figura siguiente, podríamos girar el servomotor un ángulo determinado, según el valor leído del potenciómetro.



El código necesario para ello sería el siguiente:

```
#include <Servo.h>
Servo miservo;
int valorPot = 0;
void setup() {
     miservo.attach(3);
}
void loop() {
     valorPot = analogRead(0);
/*Los valores de analogRead() van desde 0 a 1023 y los valores
```

```
aceptados por miservo.write() van desde 0 a 180, por eso reajustamos
el valor leído para poderlo utilizar con el servomotor. En este caso
particular, como los mínimos de ambos rangos son 0, en vez de map()
podríamos haber escrito una simple regla de proporcionalidad,
mediante la fórmula valorPot=valorPot*(180.0/1023.0)*/
     valorPot = map(valorPot, 0, 1023, 0, 180);
     miservo.write(valorPot);
     delay(15);  //Para dar tiempo al servo a moverse
}
```

Breve nota sobre los “softpots” o potenciómetros de “membrana”:

Un “softpot” o potenciómetro de “membrana” es un tipo de potenciómetro en forma de tira muy delgada. Si a lo largo de esa tira se ejerce presión, la resistencia cambia prácticamente de forma lineal desde unos 100 ohmios hasta unos 10 KΩ. Esto permite calcular con gran precisión la posición relativa en la tira de, por ejemplo, nuestro dedo. Por ello, estos dispositivos se suelen utilizar mucho en aparatos domésticos, como reproductores “mp3”, televisores, etc. Un ejemplo de “sofpot” es el producto nº 8607 de Sparkfun.

Estos componentes disponen de tres pines: el pin de más a la derecha (mirándolo de frente con los pines en la zona inferior) suele ser el de la alimentación, el de más a la izquierda es el de tierra y el central se corresponde a la señal obtenida como lectura analógica. Si se presiona en la zona superior de la tira, obtendremos por ese pin central una lectura de 0, y si vamos presionando hacia abajo llegaremos a obtener una lectura máxima de 1023.

Ejemplo de uso de joysticks como entradas analógicas

Un joystick internamente no es más que un conjunto de dos potenciómetros que permiten medir el movimiento de la palanca a lo largo del eje X y el eje Y (es decir, en 2 dimensiones). Por tanto, las conexiones necesarias para utilizar un joystick con nuestra placa Arduino son las siguientes: un extremo de cada potenciómetro ha de estar conectado a la fuente de alimentación del circuito (normalmente, los 5 V proporcionados por la placa Arduino), el otro extremo de cada potenciómetro ha de estar conectado a tierra, y la patilla central de cada potenciómetro ha de estar conectado a una entrada analógica diferente.

La mayoría de veces, el joystick también incorpora un pulsador interno, que se activa al apretarlo. En estos casos, para detectar estas pulsaciones generalmente deberemos conectar además otro cable a nuestra placa Arduino, pero esta vez a una entrada digital.

Para saber el desplazamiento realizado en un eje o en otro, deberemos consultar el valor leído en la entrada analógica correspondiente (el cual puede ir desde 0 en un extremo hasta 1023 en otro). Dependiendo de la magnitud de esos dos valores leídos, mediante "ifs" o "switchs" podremos decidir entonces qué hacer.

IteadStudio distribuye un módulo, el "Joystick breakout module", cuya conexión es muy sencilla. Tan solo ofrece 5 pines: "+" (a conectar a 5 V), "G" (a conectar a tierra), "X" (a conectar a una entrada analógica), "Y" (a conectar a la otra entrada analógica) y "B" (a conectar a una entrada digital, para detectar pulsaciones). Sus dos potenciómetros internos son de 10 KΩ cada uno. Otro producto parecido es el nº 27800 de Parallax, el cual, no obstante, carece del pulsador integrado; con él las conexiones a realizar son: pines "L/R+" y "V/D+" a alimentación de 5V, pin "L/R" a una entrada analógica, "V/D" a la otra entrada analógica y "GND" a tierra.

También podemos utilizar shields que incluyen un joystick y varios pulsadores extra más. Se debe consultar la documentación de cada shield para saber a qué entradas analógicas y digitales está vinculado tanto el joystick como los diversos pulsadores incluidos, de manera que podamos escribir nuestros sketches correctamente. Ejemplos de este tipo de shield son: el "InputShield" de LiquidWare, el "Arduino Input Shield" de DFRobot, el "Joystick Shield" de IteadStudio, el "Joystick Shield" de Elecfreaks o el "Joystick Shield Kit" de Sparkfun (aunque este último, tal como su nombre indica, viene en forma de kit, por lo que es necesario soldar sus partes).

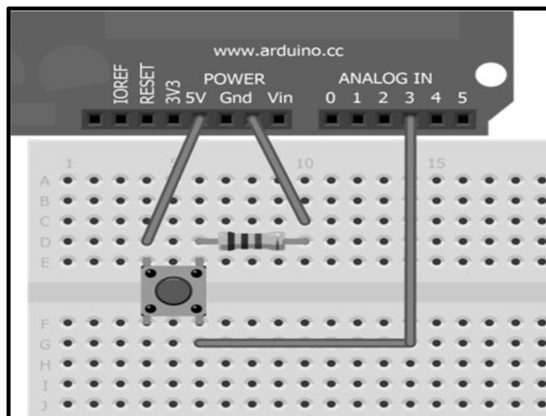
A continuación, se muestra un código muy sencillo válido para todos los módulos y shields anteriores, donde, mientras se manipula un joystick, se va visualizando en tiempo real por el "Serial monitor" los valores leídos por las entradas analógicas nº 0 (eje X) y nº 1 (eje Y).

```
byte ejeX=0;                    //Entrada analógica eje X (n° 0)
byte ejeY=1;                    //Entrada analógica eje Y (n° 1)
byte boton=3;                   //Entrada digital botón (n° 3)
int  valorx,valory,valorboton;  //Valores leídos
```

```
void setup() {
	Serial.begin(9600);
}
void loop(){
	valorx = analogRead(ejeX);
	Serial.print( "X:" );
	Serial.print(valorx);
/*Es necesario hacer una pequeña pausa entre lecturas de diferentes pines analógicas porque si no puede ser que obtengamos la misma lectura dos veces (debido al funcionamiento interno de esos pines) */
	delay(100)
	valory = analogRead(ejeY);
	Serial.print ( " | Y:" );
	Serial.print (valory);
	delay(100);
	valorboton = digitalRead(boton);
	Serial.print ( " | Botón: " );
	Serial.print (valorboton);
}
```

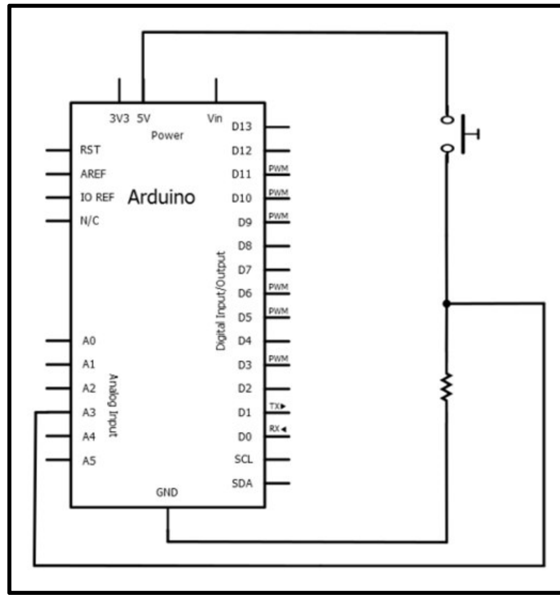
Ejemplo de uso de pulsadores como entradas analógicas

Los pulsadores se pueden utilizar para detectar valores diferentes más allá de los simples HIGH y LOW, pero para ello deberemos conectarlos a entradas analógicas. Montaremos el siguiente circuito (usando la configuración con resistencia "pull-down"):



Como se puede ver, es muy parecido al que ya vimos cuando tratamos las entradas digitales, solo que ahora el "cable de control" del pulsador está conectado a

una entrada analógica de la placa (concretamente, la número 3) en vez de a un pin digital. El esquema eléctrico correspondiente sería:



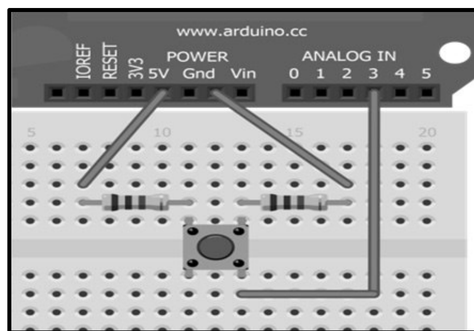
Para probar el circuito anterior, podemos utilizar un sketch muy parecido al que vimos cuando introdujimos por primera vez las entradas digitales, pero ahora estaremos viendo lo que ocurre en una entrada analógica.

```
int estadoBoton=0; //Guardará el estado "analógico" del botón
void setup(){ Serial.begin(9600); }
void loop() {
        estadoBoton=analogRead(3);
       Serial.println(estadoBoton);
       delay (50);  //Para mayor estabilidad entre lecturas
}
```

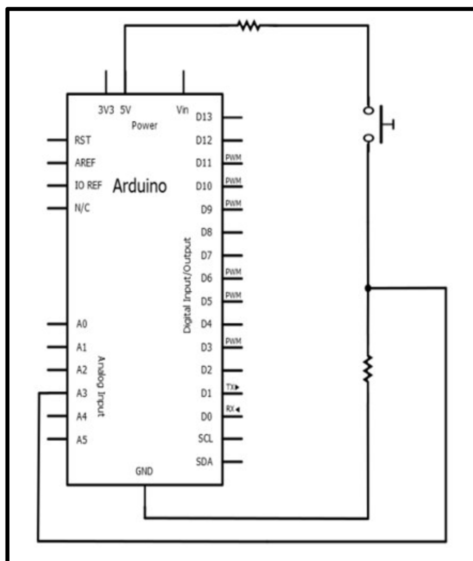
¿Y qué es lo que se ve? Que en vez de obtener un valor HIGH (1) cuando se pulsa el botón y un valor LOW (0) cuando no, en el primer caso se obtiene un valor 1023 (es decir, el equivalente, tras la conversión analógico-digital interna) a 5 V, y en el segundo un valor 0 (0 V). ¿Por qué? Porque cuando se pulsa el botón el voltaje existente entre la fuente de alimentación (los 5 V proporcionados por la placa) y el pin de entrada analógico es precisamente 5 V, ya que hay un camino directo entre ambos extremos sin ninguna caída de potencial entremedio. Pero cuando el botón

está abierto, el pin analógico está conectado a tierra a través de la resistencia "pull-down".

Si ahora queremos leer un valor de voltaje entre 0 y 5 V (¡es la gracia de tener lecturas analógicas!), deberíamos introducir un divisor de tensión en el camino entre la fuente de alimentación y el pin analógico de entrada. Concretamente, colocaremos una resistencia entre el pin de 5 V de la placa y la patilla del pulsador conectado a ella, tal como se muestra en las siguientes figuras. El valor de la resistencia puede ser cualquiera, pero cuanto mayor sea, mayor caída de potencial provocará entre sus bornes, y por tanto, menor será la lectura que reciba el pin analógico cuando se pulse el botón. Así pues, tenemos:



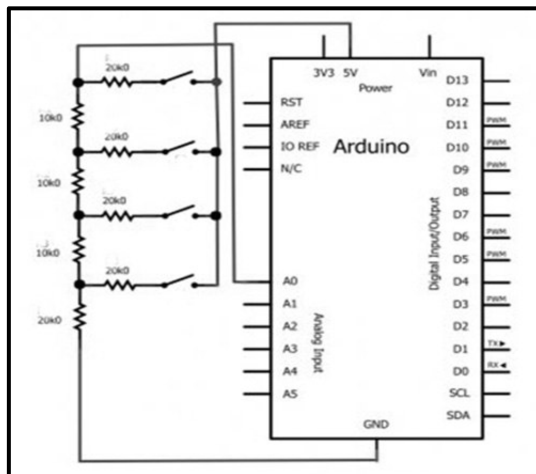
Es decir:



En el caso de haber utilizado la configuración del pulsador con resistencias “pull-up”, entonces deberíamos haber colocado el divisor de tensión entre tierra y la patilla del pulsador conectada a esta.

¿Qué ocurre ahora si volvemos a ejecutar el mismo sketch de antes? Que ahora veremos un valor intermedio entre 0 y 1023, dependiendo del valor de la resistencia que hayamos puesto como divisor de tensión (recordemos, a mayor resistencia, el valor leído será menor). Ahora mismo, por tanto, tenemos un circuito que recibe una señal analógica (de un valor fijo) solo cuando se mantiene pulsado el botón.

¿Para qué nos puede servir esta funcionalidad de los pulsadores? Para por ejemplo poder implementar un circuito R-2R “en escalera”, tal como el mostrado en la página siguiente. Este tipo de circuito utiliza varios pulsadores, todos ellos conectados a un único pin de entrada analógica. Cada uno de estos pulsadores cuando es presionado enviará una señal analógica de un determinado voltaje a la entrada a la que están conectado (en este caso, la nº 0). La gracia está en que dependiendo del pulsador que presionemos, el valor del voltaje de la señal recibida por el pin analógico será diferente. Incluso se pueden pulsar varios botones a la vez, y esto generará otro voltaje diferente (aunque es posible que algún conjunto concreto de pulsaciones se solape con otro). Todo esto es gracias al diseño R-2R “en escalera” de resistencias “pull-down” y divisores de tensión, tal como el mostrado en la figura siguiente. Este diseño se llama así porque las resistencias “pull-down” (mostradas en la figura verticalmente) tienen siempre la mitad de valor (en este ejemplo, 10 KΩ) que los divisores de tensión (mostrados en la figura horizontalmente, y, en este ejemplo, con un valor de 20 KΩ). Es preferible que todas las resistencias tengan un 1% de tolerancia para mejorar la exactitud de las medidas.



Una vez realizado el circuito R-2R, lo único que tendríamos que hacer para aprovechar estas diversas entradas es escribir un sketch que fuera comprobando (con varios "ifs" o un "switch") qué valor de voltaje recibe la placa, para así responder según lo que proceda. De esta manera, podríamos implementar un "keypad" artesano con solo un pin de nuestra placa.

De hecho, una plaquita que implementa esta misma idea es la llamada "ADKeyboard Module" de DFRobot. Esta plaquita contiene 5 botones que tan solo requieren el uso de una entrada analógica de nuestra placa Arduino para ser monitorizados de forma independiente. En la página web de este producto se ofrece además un código Arduino de ejemplo para saber cómo se puede realizar la gestión de las pulsaciones.

Otra plaquita similar es la "TWI Keyboard" de Akafugu. No obstante, esta plaquita se comunica con nuestra placa Arduino vía I^2C y por tanto, requiere 2 cables (además del de alimentación y tierra). También dispone de 5 botones independientes, los cuales pueden ser controlados mediante la librería descargable de https://github.com/akafugu/twikeyboard. Esta librería es capaz de detectar los eventos de pulsación, soltado y repetición de los botones, y gestiona correctamente el efecto de "bounce".

También podemos utilizar el producto nº 27801 de Parallax, el cual es un pulsador de cinco posiciones. Es decir, funciona como si fuera una plaquita de cinco pulsadores, aunque a simple vista se asemeje a un joystick. Su conexión es sencilla: el pin marcado como "VCC" va a 5 V, el "GND" va a tierra y los otros cinco pines se han de conectar a cinco entradas digitales de la placa Arduino. Cada uno de estos pines se corresponde con una posición del pulsador (arriba, abajo, derecha, izquierda, centro), y cuando el pulsador se coloque en una de ellas, el pin correspondiente recibirá una señal LOW (debido a las resistencias "pull-up" internas de la plaquita).

Sensores capacitivos

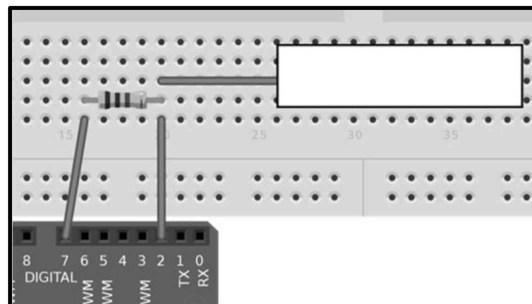
No solamente los condensadores propiamente dichos disponen de capacidad eléctrica (es decir, de posibilidad de almacenar carga): cualquier objeto sometido a un determinado voltaje y formado internamente por dos superficies conductoras separadas por un material aislante y situadas a una distancia una de la otra relativamente corta ya puede actuar como un condensador. El valor concreto de la capacidad intrínseca de ese objeto depende del tipo de material aislante utilizado y de la distancia entre ambas superficies conductoras. Concretamente, la capacidad es inversamente proporcional a esta distancia; es decir: si la distancia entre ellas

aumenta, la capacidad del componente disminuye, y viceversa. Esta variación de la capacidad puede ser utilizada para implementar un "sensor capacitivo" que sirva para deducir la distancia entre ambas superficies.

Lo interesante del asunto es que nuestro cuerpo humano actúa como una superficie conductora conectada a tierra. Esto implica que si acercamos un dedo a una plancha de metal o de cualquier otro material conductor sometido a una determinada tensión y protegido por un material aislante (como el plástico, la cerámica o la madera) estaremos creando un condensador, cuya capacidad dependerá del material aislante (normalmente fijo) y de la distancia de nuestro dedo a esa plancha (variable).

Podemos pues utilizar nuestra placa Arduino como un sensor capacitivo, detectando la aproximación o contacto de un dedo (por ejemplo) a una determinada superficie en un rango de unos pocos milímetros. Si utilizamos varios sensores, podríamos establecer diferentes reacciones en nuestro circuito, lo que nos permitiría interactuar con la placa Arduino de una manera similar a como lo hacemos con un keypad o un conjunto de pulsadores. De hecho, esta técnica es la que se utiliza habitualmente en muchos productos electrónicos con pantallas táctiles de contacto a baja presión.

Para ello, deberemos construir un circuito tan sencillo como el siguiente:



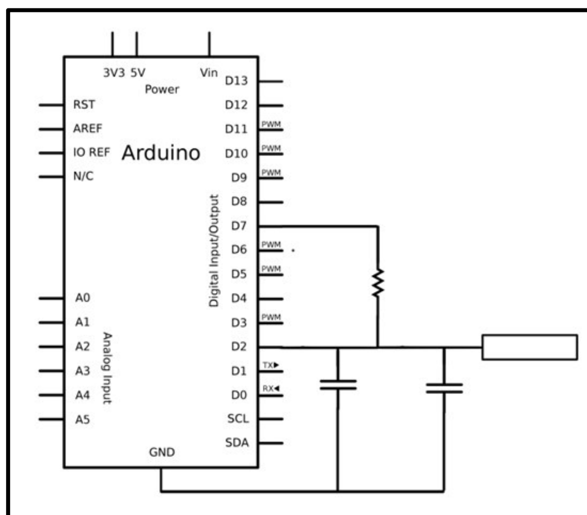
donde el rectángulo blanco representa la superficie conductora (una opción podría ser una tira de papel de aluminio, por ejemplo) preferiblemente bajo una capa aislante (una opción podría ser un simple papel).

La resistencia ha de tener un valor relativamente elevado (entre 100 KΩ y 50 MΩ son valores adecuados). Si se utiliza un valor menor de 1 MΩ, el sensor tan solo detecta la presencia de un dedo cuando existe contacto físico. Con 10 MΩ el sensor detecta la presencia de un dedo a una distancia de 5-10 cm, y con una resistencia de

40 MΩ el dedo es detectado a una distancia de 30-60 cm (dependiendo de las dimensiones de la superficie conductora). Este incremento de sensibilidad a mayores resistencias tiene el inconveniente de que la respuesta del sensor es bastante más lenta.

El circuito anterior se basa en que un pin de la placa Arduino actúe como salida digital (en el diagrama anterior sería el pin nº 7) y el otro pin (conectado a la superficie sensible) actúe como entrada digital (en el diagrama anterior sería el pin nº 2). El método de medición consiste en establecer un nuevo valor al pin de salida (HIGH o LOW) y esperar un determinado tiempo hasta que el pin de entrada cambie automáticamente al mismo estado que el pin de salida. Dependiendo del tiempo transcurrido hasta ese cambio de estado, se puede medir la capacidad generada por el contacto humano. Concretamente, el retraso entre el cambio de estado del pin de salida y el del pin de entrada está definido por la llamada "constante de tiempo", calculada mediante la expresión R·C, donde R es el valor de la resistencia y C es la capacidad presente en el pin de entrada, la cual depende de todas las otras capacidades presentes del circuito (como la resultante de la interacción humana, que es la única que varía).

La experiencia indica que es conveniente añadir al circuito anterior dos condensadores extra para mejorar la estabilidad de las lecturas realizadas. Concretamente, es recomendable conectar un condensador de 0,1 µF entre el pin de entrada y tierra, y otro pequeño condensador (de entre 0,02 y 0,4 µF) cerca de la superficie conductora, en paralelo con el generado por el cuerpo humano. Es decir, que el montaje más conveniente es el siguiente:



También es recomendable para mejorar la estabilidad de las lecturas conectar la placa Arduino a una tierra externa de referencia, o bien, si está conectada a un computador, que este esté enchufado a la red eléctrica.

Afortunadamente, no tenemos que tener grandes conocimientos de electricidad ni de condensadores para poder interpretar las lecturas obtenidas por el circuito anterior, porque la librería semi-oficial “CapacitiveSensor”, descargable de http://www.arduino.cc/playground/Main/CapacitiveSensor, nos facilitará muchísimo la faena. Las funciones más importantes de esta librería son:

CapacitiveSensor(): esta función se ha de escribir en la zona de declaraciones globales del sketch, fuera por tanto de las funciones “setup()” y “loop()”. Su primer parámetro es el número del pin de la placa Arduino que actuará como salida y su segundo parámetro es el pin de entrada. Devuelve un objeto de tipo CapacitiveSensor (lo llamaremos “micapsense”) que representa el sensor y que nos permitirá utilizar el resto de funciones de la librería. Si tenemos varios sensores, cada uno de ellos será representado por un objeto CapacitiveSensor diferente. Un ejemplo de declaración-creación de un objeto CapacitiveSensor sería: `CapacitiveSensor micapsense = CapacitiveSensor(4,2);`

micapsense.capacitiveSensor(): esta función requiere un único parámetro –de tipo “byte”–, que es el número de muestras (30 sería un buen valor) que se quieren obtener de la superficie conductora, para realizar el cálculo pertinente y devolver (en forma de valor de tipo “long”) la capacidad medida, en unidades arbitrarias. Esta función tiene en cuenta la capacidad de base siempre presente en el circuito y la resta de la capacidad detectada variable, por lo que normalmente devuelve un valor bajo cuando no hay contacto o proximidad humana.

micapsense.set_CS_Autocal_Millis(): esta función requiere un único parámetro –de tipo “unsigned long”– que indica cada cuántos milisegundos la librería recalculará y recalibrará la capacidad de base constante en el circuito que se resta en las medidas de las variaciones detectadas. El valor por defecto son 20 segundos. La recalibración se puede desactivar escribiendo el valor "0xFFFFFFFF" como valor del parámetro de esta función.

Un ejemplo sencillo demostrativo de su uso es el siguiente sketch. Para probarlo, se puede acercar y alejar una mano de alguno de los sensores para comprobar en el “Serial monitor” los cambios en los valores detectados.

```
#include <CapacitiveSensor.h>
/*Declaro tres sensores capacitivos,
todos ellos con el pin n° 4 como pin de salida */
CapacitiveSensor   micapsense1 = CapacitiveSensor(4,2);
CapacitiveSensor   micapsense2 = CapacitiveSensor(4,6);
CapacitiveSensor   micapsense3 = CapacitiveSensor(4,8);
void setup(){
   //Desactivo la calibración automática, solo como ejemplo
   micapsense1.set_CS_AutocaL_Millis(0xFFFFFFFF);
   Serial.begin(9600);
}
void loop(){
    long inicio, total1,total2,total3;
    inicio = millis();
    total1 =  micapsense1.capacitiveSensor(30);
    total2 =  micapsense2.capacitiveSensor(30);
    total3 =  micapsense3.capacitiveSensor(30);
    //Compruebo el rendimiento
    Serial.println(millis() - inicio);
    //Muestro el valor obtenido por los sensores
    Serial.println(total1);
    Serial.println(total2);
    Serial.println(total3);
    delay(10);  //Me espero para darle tiempo al canal serie
}
```

Tal como se ve en el código anterior, si queremos tener varios sensores (es decir, varias superficies conductoras), para distinguir diferentes contactos, no es necesario utilizar para cada sensor dos parejas de pines: el pin de salida puede ser compartido. Así, podríamos tener como único pin de salida por ejemplo el nº 7 conectado mediante la resistencia adecuada a un pin de entrada cualquiera (conectado a su vez a la superficie conductora), y tenerlo además conectado mediante otra resistencia, en paralelo a la anterior, a otro pin de entrada diferente, y tenerlo conectado también mediante otra resistencia, en paralelo a las anteriores, a otro pin de entrada, y así.

De todas maneras, si no queremos construir desde cero un circuito con varios sensores capacitivos, podemos utilizar shields que ya nos los incorporan de fábrica cómodamente. Por ejemplo, Modern Device distribuye un shield llamado "ArduCapSense" que está fabricado utilizando los principios básicos explicados en los párrafos anteriores y que, por tanto, es programable con la misma librería

"CapacitiveSensor". Dispone de ocho botones (sensores capacitivos) y además, dispone de una salida jack de audio y de varios LEDs por si se quiere identificar el botón pulsado de forma acústica o lumínica respectivamente; en la página del producto se pueden encontrar códigos de ejemplo para ambos casos. No obstante, el shield se distribuye en forma de kit, por lo que se precisa soldar los componentes.

Sparkfun distribuye su "Touch shield" (nº de producto 10508), que ofrece nueve botones (como un "keypad" estándar), pero utiliza un chip especializado, el sensor de contacto capacitivo MPR121 de Freescale. Esto hace que la librería "LibCapSense" no nos sea útil; sin embargo, en la página del producto se puede descargar un código Arduino de ejemplo para comprender su manejo (además de la librería particular de la que este hace uso). El mismo sensor capacitivo también se distribuye en forma de plaquita breakout (nº de producto 9695), a la cual se le pueden conectar hasta 12 sensores capacitivos ("botones") externos. Esta plaquita funciona a 3,3 V y se comunica a través del protocolo I^2C con el exterior, por lo que para utilizarla con nuestra placa Arduino es necesario utilizar un convertidor de nivel y para programarla es necesario utilizar la librería oficial Wire (en la página del producto se ofrecen diversos códigos de ejemplo). Otra placa muy similar a la anterior pero incluyendo dentro de sí los 12 sensores capacitivos en forma de "keypad" ya prefabricado es el producto 10250 de Sparkfun.

Cambiar el voltaje de referencia de las lecturas analógicas

Ya comentamos en el capítulo 2 el concepto de "voltaje de referencia". Básicamente, es un valor que define la precisión con la que se convierte una señal analógica (leída en una entrada analógica con *analogRead()*) a una señal digital equivalente (transformada por el conversor interno de la placa Arduino). Cuanto menor sea ese voltaje de referencia, mayor es la exactitud en la conversión. Por defecto, este valor es de 5 V, pero si la precisión que nos ofrece no nos es suficiente (ya que con 5 V de referencia se llega a una distancia entre valores digitales contiguos de aproximadamente 4,9 mV), lo podemos reducir haciendo uso de la función *analogReference()* y, opcionalmente, del pin AREF de la placa.

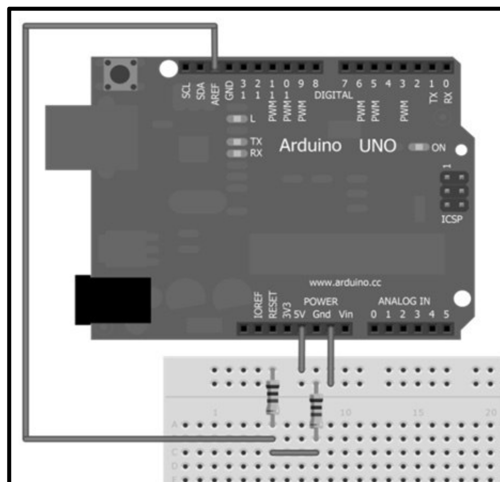
analogReference(): configura el voltaje de referencia usado para la conversión interna de valores analógicos en digitales. Dispone de un único parámetro, que en la placa Arduino UNO puede tener los siguientes valores: la constante predefinida DEFAULT (que equivale a establecer el voltaje de referencia es 5 V –o 3,3 V en las placas que trabajen a esa tensión–, el cual es el valor por defecto), o la constante predefinida INTERNAL (donde el voltaje de referencia entonces es de 1,1 V) o la constante predefinida EXTERNAL

(donde el voltaje de referencia entonces es el voltaje que se aplique al pin AREF –“Analogue REFerence”–). Es muy importante ejecutar siempre esta función antes de cualquier lectura realizada con *analogRead()* para evitar daños en la placa. Esta función no tiene valor de retorno.

Si usamos la opción de utilizar una fuente de alimentación externa para establecer el voltaje de referencia, deberemos conectar el borne positivo de esa fuente al pin-hembra AREF de la placa y su borne negativo a la tierra común. Para no dañar la placa, es muy importante que el voltaje de referencia externo aportado al pin AREF no sea nunca menor de 0 V (es decir, invertido de polaridad) ni mayor de 5 V.

Rebajar el voltaje de referencia es útil cuando sabemos que los valores de las entradas analógicas no alcanzan nunca los 5 V. Por ejemplo, si sabemos que el valor máximo de una determinada señal de entrada es de 2,5 V, podríamos rebajar con una fuente adecuada el voltaje de referencia a 2,5 V, para hacer así que cada valor digital se correspondiera con su valor analógico respectivo en una precisión de 2,5V/1024≈2,5 mV, doblando pues así la resolución utilizada. Hay que tener en cuenta, no obstante, que los efectos del ruido aumentan a medida que aumentemos la resolución en los valores leídos.

Podemos construir una fuente de referencia externa fácilmente (¡y muy barata!) con un simple divisor de tensión. Por ejemplo, si queremos que el voltaje de referencia sea de 2,5 V, podemos utilizar los 5 V que aporta el pin “5V” de la propia placa Arduino para reducirlo mediante dos resistencias idénticas conectadas en serie (no importa su valor numérico concreto, lo que importa es que sean iguales) y utilizar el voltaje presente entre las dos (reducido a 2,5 V por simple Ley de Ohm) como entrada del pin AREF. Es decir, diseñar un circuito como este:



Se deberían usar resistencias de baja tolerancia (1%) para que la señal de referencia sea lo suficientemente precisa. Aunque la manera más exacta de proporcionar una señal rebajada y estable sería usando un regulador de tensión apropiado.

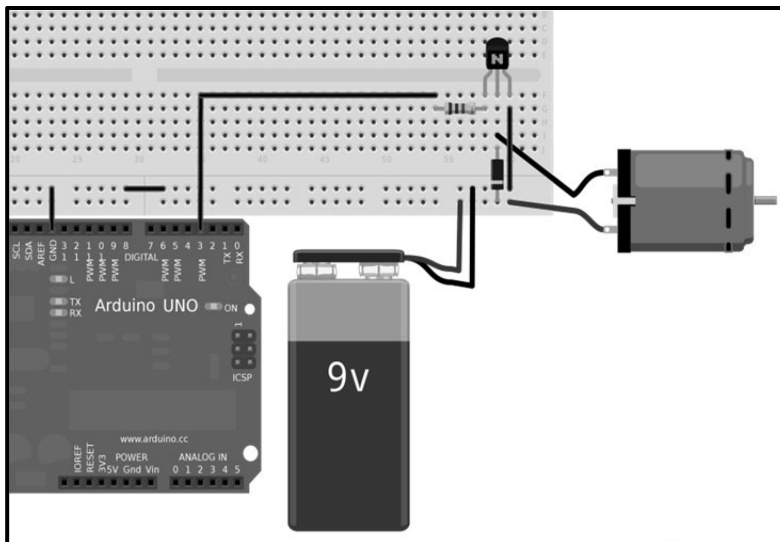
El siguiente código muestra cómo utilizar un voltaje de referencia externo. Necesitamos un circuito con un potenciómetro, la patilla central que ha de estar conectada al pin analógico de entrada nº 2. Supondremos que la fuente externa de referencia proporciona por ejemplo 1,8 V (y por tanto, dividiendo entre 1024 tendríamos 1,75 milivoltios de resolución entre cada lectura de *analogRead()*).

```
int valorPot=0;
float voltajePot=0;
void setup(){
     analogReference(EXTERNAL); //Ésta es la línea clave
     Serial.begin(9600);
}
void loop(){
     valorPot=analogRead(2);
//Primero se muestra la lectura en forma de porcentaje
     Serial.println((voltajePot/1024)*100);
/*La siguiente línea convierte el valor de valorPot en un valor de
voltaje. Fijarse que los valores obtenidos de analogRead() van desde
0 a 1023 y los valores que queremos van desde 0 al voltaje de
referencia (1,8V hemos supuesto en este ejemplo). Podríamos pensar en
utilizar la función map(), pero esta solo devuelve valores enteros,
por lo que no nos serviría demasiado. No obstante, como en este caso
particular los mínimos de ambos rangos son 0, en vez de map() podemos
escribir una simple regla de proporcionalidad. Hay que tener en
cuenta el detalle de haber especificado los valores numéricos de la
fórmula como de tipo "float" (añadiéndoles el 0 decimal) para que el
resultado obtenido sea de tipo "float" también y no se trunque.*/
     voltajePot=valorPot*(1.8/1023.0); //En V
     voltajePot=voltajePot*1000; //Lo convertimos a mV
     Serial.println(voltajePot);
     delay(100);
}
```

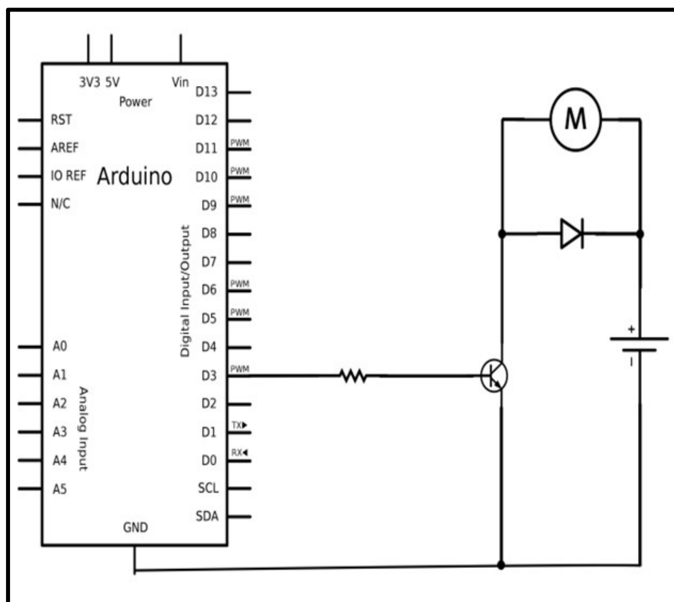
CONTROL DE MOTORES DC

En el apartado del capítulo anterior correspondiente a los motores no hablamos de cómo escribir código para controlar motores DC. Esto es porque no

existe una librería Arduino específica para ellos, ya que estos dispositivos se pueden controlar con simples señales analógicas. Cuanto mayor sea el valor de la salida PWM enviado al motor, a más voltaje estará sometido y, por tanto, más rápido girará. Así de simple. Probaremos este funcionamiento con un circuito tal como el siguiente:



Tal vez su esquema eléctrico sea más claro:



Lo primero que llama la atención es la necesidad de utilizar una alimentación externa (en este caso, una pila de 9 V) porque los motores por lo general realizan un elevado consumo, y con los 40 mA y 5 V que ofrecen los pines de la placa Arduino no es suficiente. Lo segundo que llama la atención es el transistor. En este circuito lo utilizamos como simple válvula accionada por la señal PWM proveniente de la placa. Si esta vale 0, el transistor mantendrá abierto el circuito del motor y este no girará. A medida que se vaya aumentando el valor de voltaje PWM enviado a la base del transistor, por esta circulará más corriente. Esto provocará que entre el colector y el emisor vaya pasando más y más corriente (o dicho de otra forma, que se vaya reduciendo la resistencia entre estos dos puntos más y más). Esto provocará a su vez que el motor vaya siendo sometido a un mayor potencial y por tanto, que vaya girando más rápido. El caso extremo aparece cuando el transistor actúa como un simple circuito cerrado; esto ocurre al llegar a un determinado valor de la señal PWM, el cual provoca la llamada "intensidad de saturación" en la base del transistor. En ese momento, la resistencia entre colector y emisor es nula, por lo que el motor se somete a la tensión íntegra ofrecida por la fuente, y, por tanto, el motor gira a la máxima velocidad.

Está claro que si en vez de haber sometido el transistor a una señal PWM, lo sometiéramos a una señal digital, el transistor actuaría como un interruptor, parando o moviendo el motor según si recibe una señal LOW o HIGH, respectivamente.

Es importante aclarar que en el esquema hemos utilizado un transistor NPN de tipo Darlington modelo TIP120 (otro similar sería el BD645). El TIP120 puede trabajar sometido a tensiones entre el colector y el emisor (V_{ce}) de hasta 60 V pero admite una corriente por el colector (I_c) de hasta tan solo 5 A. Por otro lado, hay que tener en cuenta la ubicación de sus patillas para comprender el diseño del circuito: mirándolo de frente, la base es su patilla izquierda, el colector es su patilla central y el emisor es su patilla derecha; en otro modelo de transistor las patillas pueden estar intercambiadas. A destacar también el divisor de tensión (2 KΩ sería un buen valor) conectado en serie a su base.

El mismo circuito lo podríamos haber diseñado utilizando un transistor de tipo MOSFET (por ejemplo, el RFP30N06LE). En ese caso, la única diferencia es sustituir la resistencia en serie del circuito anterior por una resistencia "pull-up" (de 10 KΩ por ejemplo) situada entre el pin de salida PWM de la placa Arduino y tierra. Y ya está. La ventaja de utilizar este transistor es que, aunque su V_{ce} máximo admitido sigue siendo de 60 V, la I_c máxima es de 30 A, pudiendo ser usado por tanto en circuitos con mucho mayor consumo. Podemos utilizar otros transistores diferentes (incluso un optoacoplador), pero en todo caso deberemos comprobar en su

datasheet que sus V_{ce} e I_c máximos sean valores dentro del rango de trabajo del circuito.

Por otro lado, la resistencia que conectemos en serie a la base del transistor (necesaria para no dañarla) debe ser de un valor lo suficientemente pequeño como para poder alcanzar la corriente mínima con la que se llega al estado de saturación del transistor. El valor adecuado de esta resistencia se puede calcular mediante la expresión $(V_{pin} – 0,7)/I_{bsat}$, donde V_{pin} es la tensión que proporciona el pin de la placa Arduino donde está conectada la base (en nuestro caso, es siempre 5 V), 0,7 V es la caída de tensión típica entre la base y el emisor del transistor (aunque se puede mirar en el datasheet del transistor como V_{be}) e I_{bsat} es precisamente la corriente mínima con la que se llega al estado de saturación del transistor, más allá de la cual no se obtiene más amplificación en I_c. Si I_{bsat} no nos lo proporcionaran en el datasheet, se podría calcular fácilmente a partir de la expresión $I_{bsat} = I_{cmax}/h_{FE}$, donde tanto I_{cmax} como h_{FE} sí que son consultables en el datasheet (h_{FE} es la llamada ganancia de corriente del transistor: si hay varios valores se ha de elegir el más pequeño).

Lo importante del circuito anterior es fijarse en que hemos utilizado un transistor para “desacoplar” dos circuitos que funcionan a diferentes voltajes para que no haya peligro de dañarlos. Es decir, los 9 V utilizados para alimentar el motor no son nunca percibidos por la placa Arduino, la cual continúa trabajando como siempre a 5 V. El transistor hace pues de barrera entre los dos circuitos, de tal forma que uno (sometido a 5 V) simplemente sirve para controlar el funcionamiento del otro (sometido a 9 V) pero sin peligro para el primero. Si se desea una protección algo más sofisticada, se puede sustituir el transistor por un optoacoplador, pero en el circuito anterior esto no es necesario.

Es interesante conocer la potencia consumida por el transistor en un circuito como este, calculable mediante la expresión $P = I_c·V_{CE}$. En el modo de corte, $I_c = 0$, por lo que la potencia consumida es 0, y en el modo de saturación $V_{CE} = 0$ (aproximadamente), por lo que la potencia también es prácticamente nula. Esto quiere decir que el transistor no debería calentarse en ningún rango de su uso y no se necesita tener en cuenta ninguna potencia máxima de trabajo.

Por último, la presencia del diodo tiene una razón: hace de diodo “fly-back”. Ya hemos comentado en el capítulo anterior que cuando se deja de alimentar una bobina (y los motores están hechos de bobinas) durante unos milisegundos entre los bornes del motor aparece un voltaje de hasta varios centenares de voltios polarizado a la inversa del de la fuente. Cuando hay alimentación, el diodo está situado “al revés” y no hace nada, pero cuando aparece ese voltaje inverso, el diodo permite que

la corriente generada retorne al motor y no se dirija al transistor (ya que si no, lo freiría). Necesitamos por tanto un diodo lo suficientemente rápido para reaccionar al voltaje inverso y lo suficientemente fuerte para resistirlo: el modelo 1N4001 o SB560 son buenas opciones.

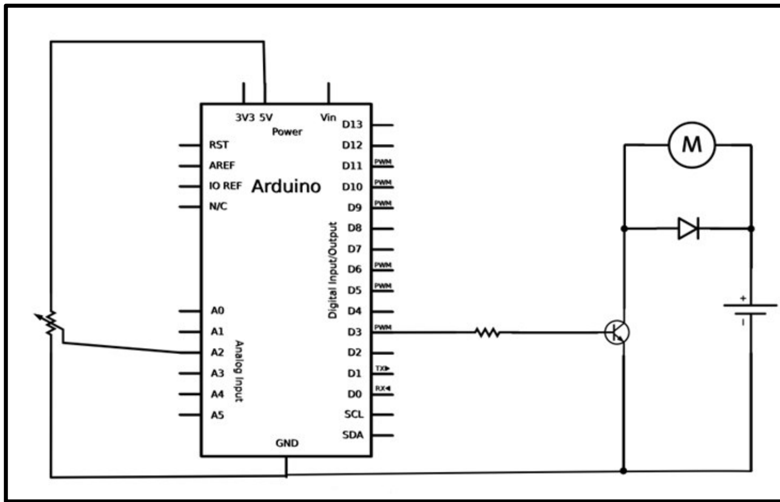
Un error común en el principiante es olvidarse de conectar el emisor del transistor, además de al polo negativo de la fuente, al pin GND de la placa. Esta conexión es necesaria porque que la señal PWM necesita un camino de retorno.

Otro detalle que debemos tener en cuenta es que aunque podemos considerar que el montaje anterior se compone de dos circuitos separados por el transistor (el circuito del motor y el circuito de la señal PWM), estos están interrelacionados, por lo que es fundamental que las tierras de ambos sean comunes. Es por eso que el polo negativo de la fuente está conectado al pin GND de la placa Arduino.

Ejemplo 6.26: Aquí vemos un sketch que aumenta la velocidad de giro del motor hasta llegar a su máximo y seguidamente la disminuye hasta pararse, y volver a acelerarse otra vez, y así siempre:

```
void setup(){
  pinMode(3, OUTPUT);
}
void loop(){
  int i;
  for(i = 0; i<255; i++){
    analogWrite(3,i);
    delay(15);
  }
  for(i = 255; i>=0; i--){
    analogWrite(3,i);
    delay(15);
  }
}
```

Añadiremos ahora al circuito anterior un potenciómetro (en el pin de entrada analógica nº 2, por ejemplo), de tal manera que la velocidad de giro del motor venga ahora dada por el valor leído de la resistencia variable. El esquema eléctrico del nuevo diseño ha de ser así:



Ejemplo 6.27: Para hacer el sketch algo más sofisticado, hemos añadido la condición de que si el valor leído por el potenciómetro no llega a un nivel mínimo ("umbral"), el motor no se pone en marcha; y cuando se pone en marcha, entonces ya sí que su velocidad es proporcional a la lectura recibida del potenciómetro:

```
int lectura = 0;
int umbral= 700;
void setup() {
    pinMode(3, OUTPUT);
}
void loop() {
  lectura = analogRead(2);
  if (lectura > umbral) {
  //Ajusto el valor de "lectura" para que caiga entre 0 y 255
        analogWrite(3, lectura/4);
  } else {
        digitalWrite(3, LOW);
  }
}
```

El código anterior es perfectamente válido para cualquier otra tipo de entrada analógica, como las que veremos en el capítulo siguiente. Así, un termistor o un fotorresistor se comportan de forma equivalente a un potenciómetro, por lo que podríamos diseñar circuitos que pudieran en marcha un motor cuando se detectara cierta cantidad de luz (para por ejemplo, bajar una persiana) o cuando se detectara una cierta temperatura (para por ejemplo activar un ventilador) ,etc.

No debería costar demasiado sustituir el potenciómetro del diseño anterior por un pulsador, y conseguir que el motor solamente gire cuando se mantenga apretado el pulsador. Se deja como ejercicio.

Otro ejercicio podría ser probar de enviar, tal como ya hicimos con los servomotores, un dato numérico a través del canal serie para, en este caso, especificar la velocidad a la que deseamos girar el motor. Se deja como ejercicio.

El chip L293

Desgraciadamente, los circuitos con un solo transistor ya sabemos qué carencia tienen: el motor solo puede girar en un sentido. Para que pueda girar en los dos, se necesita utilizar un “puente H” . Podemos optar por utilizar en nuestros circuitos un chip integrado como el L293, el SN754410 o el L298, conectándolos directamente a una breadboard, por ejemplo. Todos estos chips permiten controlar 2 motores DC conectando 4 salidas digitales de nuestra placa (2 para cada motor). Para hacer girar cada motor en un determinado sentido, se ha de establecer una salida a HIGH y la otra a LOW y para hacerlo girar en sentido contrario, se han de establecer al revés (LOW y HIGH). El L293 y el SN754410 tienen una distribución de patillas idéntica, pero diferente al L298. En nuestro ejemplo utilizaremos el L293, ya que su distribución es algo más sencilla.

Este chip en realidad tiene dos puentes (uno para cada motor): uno en su lado izquierdo y otro en el derecho. Puede entregar hasta 1 A por motor y opera entre 4,5 V y 36 V, así que hay que elegir un motor DC acorde con estas características. La distribución de sus pines es la siguiente:

- **Pin 1**: activa o desactiva un motor, según si recibe una señal HIGH o LOW. También sirve para especificar la velocidad de giro si recibe una señal PWM.
- **Pin 2**: envía la señal de giro (HIGH o LOW) en un sentido para un motor.
- **Pin 3**: donde se conecta uno de los dos terminales de un motor.
- **Pin 4 y 5**: tierra.
- **Pin 6**: donde se conecta el otro terminal de un motor.
- **Pin 7**: envía la señal de giro (HIGH o LOW) en el otro sentido para un motor.
- **Pin 8**: alimentación del motor (debe ser un valor suficiente).
- **Pin 9-11**: si no se usa un segundo motor, pueden estar desconectados. El nº 9 es para enviar la señal de

activación/desactivación y PWM, el nº 10 es para enviar la señal de giro en un sentido y el nº 11 es para conectar un terminal del segundo motor.

- **Pin 12 y 13**: tierra.
- **Pin 14 y 15**: si no se usa un segundo motor, pueden estar desconectados. El nº 14 es para conectar el otro terminal del segundo motor y el nº 15 es para enviar la señal de giro en sentido contrario.
- **Pin 16**: alimentación del propio chip (ha de estar conectado a 5 V).

Según las combinaciones de señales que se envíen a las diferentes patillas de este chip, el motor girará más o menos velozmente en un determinado sentido. Concretamente, si suponemos que el pin nº 1 envía una señal HIGH, cuando en el pin nº 2 haya una señal HIGH y en el pin nº 7 haya una señal LOW, el motor girará a la derecha, y cuando en el pin nº 2 haya una señal LOW y en el pin nº 7 haya una señal HIGH, el motor girará a la izquierda. Si en ambos pines hay una señal igual (HIGH o LOW), el motor se parará.

Al circuito formado por la placa Arduino, el motor y el chip L293 es recomendable añadir además un condensador de 10-100 µF conectado cerca del motor, entre la alimentación del motor (pin nº 8) y tierra. Esto suavizará los picos de voltaje que se producen cuando el motor se enciende y evitará que el microcontrolador se reinicie debido a ello: es decir, estamos hablando de un condensador “by-pass”. Cuanto mayor sea la capacidad del condensador, más carga podrá almacenar pero más tiempo necesitará para liberarla.

Podemos incluir también un pulsador conectado a algún pin digital de la placa Arduino para alternar los dos sentidos de giro del motor. Lo más sencillo es hacer girar el motor en un sentido mientras el pulsador no está apretado, y hacerlo girar en el otro cuando sí está apretado. Así es como el sketch 6.28 opera, de hecho.

Ejemplo 6.28: Una vez ya tenemos el circuito montado, el código es simple:

```
const int switchPin = 2;  //Conectado al pulsador
const int motor1Pin = 3;  //Conectado al pin n° 2 del L293
const int motor2Pin = 4;  //Conectado al pin n° 7 del L293
const int enablePin = 9;  //Conectado al pin n° 1 del L293
void setup() {
    pinMode(switchPin, INPUT);
    pinMode(motor1Pin, OUTPUT);
    pinMode(motor2Pin, OUTPUT);
```



```
    pinMode(enablePin, OUTPUT);
    digitalWrite(enablePin, HIGH);
}
void loop() {
    if (digitalRead(switchPin) == HIGH) {
      digitalWrite(motor1Pin, LOW);
      digitalWrite(motor2Pin, HIGH);
    } else {
      digitalWrite(motor1Pin, HIGH);
      digitalWrite(motor2Pin, LOW);
    }
}
```

En el código anterior no podemos variar la velocidad de giro del motor porque el pin nº 1 del L293D tan solo recibe una señal HIGH y ya está. Pero podríamos enviarle una señal de tipo PWM mediante *analogWrite()* y hacerla variar por la acción de un potenciómetro, por ejemplo. Así podríamos controlar la velocidad de giro también. Se deja como ejercicio.

Módulos de control para motores DC

Otra opción diferente para controlar motores DC, en vez de conectar el chip directamente a una breadboard, es utilizar alguna placa breakout específica. De esta forma, el cableado es menos enrevesado y el montaje es mucho más sencillo. Usando un módulo, lo único que deberemos hacer es, además de alimentarlo y enchufar el motor a sus bornes pertinentes, localizar los dos terminales de control de sentido de giro (y conectarlos a dos pines digitales de la placa Arduino), y el terminal de activación/desactivación y control de la velocidad de giro (y conectarlo a un pin de salida PWM). Y poco más.

La placa TB6612FNG

Una placa breakout recomendable es por ejemplo el producto nº 9457 de Sparkfun. Este módulo incorpora el chip TB6612FNG, el cual incorpora dos "puentes H", por lo que es capaz de manejar dos motores DC de forma independiente, soportando un consumo constante por motor de 1,2 A (con picos puntuales de 3,2 A) y de hasta 13 V de trabajo. Las conexiones a realizar son las siguientes:

Placa breakout	Exterior
VM	Borne positivo de la fuente de alimentación externa
VCC	Pin "5V" de Arduino.

Placa breakout	Exterior
GND	Borne negativo de la fuente de alimentación externa y Pin GND de Arduino
A01	Terminal del motor A
A02	Terminal del motor A
B02	Terminal del motor B
B01	Terminal del motor B
GND	-
PWMA1	Pin de salida PWM de Arduino
AIN2	Pin de salida digital de Arduino
AIN1	Pin de salida digital de Arduino
STBY	Pin de salida digital de Arduino
BIN1	Pin de salida digital de Arduino
BIN2	Pin de salida digital de Arduino
PWMB	Pin de salida PWM de Arduino
GND	-

Los pines “IN1” y “IN2” sirven para controlar el sentido de giro de los motores A y B según el caso: cuando “IN1” esté a HIGH e “IN2” a LOW, el motor girará en un sentido e, intercambiando sus valores, girará en el otro. El pin “PWM” sirve para controlar la velocidad de los motores A y B, según el caso. El pin “STBY” permite desconectar ambos motores de la fuente de alimentación si se le envía una señal de valor LOW.

Ejemplo 6.29: A continuación, se muestra un código que controla solo un motor:

```
int STBY = 10; //Salida digital donde se conecta el pin “STBT”
//Motor A: un pin controla la velocidad y otros dos el sentido
int PWMA = 3; //Salida PWM donde se conecta el pin “PWMA”
int AIN1 = 9; //Salida digital donde se conecta el pin “AIN1” int
AIN2 = 8; //Salida digital donde se conecta el pin “AIN1”
void setup(){
  pinMode(STBY, OUTPUT);
  pinMode(PWMA, OUTPUT);
  pinMode(AIN1, OUTPUT);
  pinMode(AIN2, OUTPUT);
}
void loop(){
  //Muevo el motor 1 a máxima velocidad a la izquierda
  mover(0, 255, 1);
  delay(1000);
```



```
  parar();
  delay(250);
  //Muevo el motor 1 a media velocidad a la derecha
  mover(0, 128, 0);
  delay(1000);
  parar();
  delay(250);
}
void mover(int motor, int velocidad, int direccion){
//motor: vale 0 para el motor A y 1 para el motor B
//velocidad: 0 equivale a ninguna y 255 a la velocidad máxima
//dirección: 0 es sentido de las agujas del reloj y 1 al revés
  boolean inPin1 = LOW; //Asumo un sentido de giro inicial
  boolean inPin2 = HIGH;
  digitalWrite(STBY, HIGH); //Desactivo el standby
  if(direccion == 0){
    inPin1 = LOW;
    inPin2 = HIGH;
  }
  if(direccion == 1){
    inPin1 = HIGH;
    inPin2 = LOW;
  }
  if(motor == 0){
    digitalWrite(AIN1, inPin1);
    digitalWrite(AIN2, inPin2);
    analogWrite(PWMA, velocidad);
  }
}
void parar(){
  digitalWrite(STBY, LOW); //Activo el standby
}
```

Otros módulos

Otro módulo diferente es el “Motor Driver 2A Dual L298 H-Bridge” de Sparkfun (producto nº 9670), que contiene el chip L298, permitiendo conectar dos motores DC independientes a 2A, o bien un motor DC a 4A. Otras placas breakout similares son el “L298 Compact Motor Driver” de Solarbotics, o el “DC Motor Driver Breakout” de CuteDigi o el “Two-Channel DC Motor Driver Breakout Board” de SCMDigi o el “3-Wire Robot Motor Driver Board” de Cal-Eng. Todos estos módulos disponen del chip L298 y de dos entradas digitales para controlar el sentido de giro y el freno de cada motor más otra entrada PWM más para controlar la velocidad.

Dependiendo de la ubicación de un jumper específico, pueden usar la misma alimentación para los motores y el chip (convenientemente regulada a 5 V), o bien recibir la alimentación de parte de dos fuentes externas separadas.

Cal-Eng por su parte, también ha diseñado las placas NanoDuino y Microduino, que no es más que una placa Arduino con el chip L298 integrado, a un reducido tamaño, y programable vía FTDI. Esta placa está especialmente pensada para robótica.

Shields de control para motores DC (y paso a paso)

En vez de los módulos externos vistos en el apartado anterior podemos utilizar shields específicas para el control de motores DC (los cuales suelen servir también para el control de steppers y servomotores). De esta forma, podemos ahorrar cableado y ganar espacio en nuestros proyectos: tan solo deberemos alimentar el shield con una fuente externa, conectar los terminales del motor a los bornes adecuados y nada más.

El “Adafruit Motor Shield”

Además del shield oficial de Arduino (que, recordemos, permite manejar dos motores DC o bien un motor paso a paso), un shield interesante a considerar es el “Motor/Stepper/Servo Shield”, producto nº 81 de Adafruit. Este shield tiene dos conectores para sendos servomotores (a 5 V) e incorpora además 2 chips L293D. Esto quiere decir que tiene 4 “puentes H” (cada chip L293D tiene dos), por lo que puede controlar hasta 4 motores DC bidireccionales o bien 2 motores paso a paso unipolares o bipolares (o bien una combinación de 2 motores DC y 1 motor paso a paso). Para conectar un motor DC, simplemente se deberían conectar sus dos cables a algunos de los terminales M1, M2, M3 y M4; para conectar un motor stepper bipolar, los cables de una bobina se deberían conectar al terminal M1 y los de la otra a M2 (un segundo stepper bipolar se podría conectar a los terminales M3 y M4); si el stepper fuera de tipo unipolar, se conectaría igual y el resto de cables sobrantes tendrían que ir a tierra.

Todos los pines digitales de entrada/salida (excepto el nº 2) de este shield son utilizados para la comunicación entre la placa Arduino y shield, por lo que no se podrán utilizar para otra cosa. Las 6 entradas analógicas sí que están disponibles (y se pueden usar como pines digitales también).

El chip L293D de este shield proporciona 0,6 A por puente, con un pico máximo de 1,2 A para momentos puntuales, y puede manejar motores funcionando

entre 4,5 V y 25 V. Para alimentar los motores DC y paso a paso, se necesita una fuente externa conectada al bloque de dos bornes etiquetado como "EXT_PWR", y dependiendo de si está colocado o no un jumper concreto en el shield, esta alimentación servirá también para la placa Arduino o esta deberá alimentarse separadamente vía USB u otra fuente externa independiente (que es lo recomendable).

Para manejar los servomotores conectados a este shield se puede utilizar la librería oficial Servo de Arduino, pero para controlar los motores DC y paso a paso, es necesario utilizar una librería propia de Adafruit, la "Adafruit Motor Shield Library" (https://github.com/adafruit/Adafruit-Motor-Shield-library. Desgraciadamente, no disponemos de espacio suficiente para detallar el funcionamiento de esta librería, por lo que remito a los códigos de ejemplo y a su documentación oficial (http://www.ladyada.net/make/mshield/use.html).

Existe otra librería suplementaria a la "Adafruit Motor Shield Library" que aporta un control extra avanzado en el uso de motores paso a paso, con posibilidad de aceleración y desaceleración, o con acceso concurrente a varios steppers. Esta librería se puede descargar de https://github.com/adafruit/AccelStepper, donde también se pueden consultar varios códigos de ejemplo.

En realidad, si se desea utilizar este shield tan solo para manejar motores DC, es posible no tener que utilizar la librería de Adafruit y simplemente escribir código Arduino tal cual, si se siguen las instrucciones especificadas en la siguiente página de la web oficial: http://arduino.cc/playground/Main/AdafruitMotorShield.

Otros shields

Existen más shields que incorporan la circuitería necesaria para manejar distintos tipos de motores (DC, Servo o Steppers). Por ejemplo, un shield prácticamente idéntico al de Adafruit es el "Rotoshield" de Snootlab: utiliza el mismo chip L293D, permite controlar el mismo número y tipo de motores y trabaja en los mismos rangos de consumo. Entre las diferencias podemos destacar que utiliza comunicación I^2C con la placa Arduino (por lo que libera muchos más pines de entrada/salida para poderlos utilizar en otros menesteres), que ofrece hasta ocho salidas PWM extras, y que acepta una alimentación externa de hasta 18 V. Para programarla se necesita utilizar la librería llamada "Snootor", disponible en https://github.com/Snootlab/Snootor.

Un shield que utiliza el chip L298N (como el del shield oficial de Arduino) es el "Motomama" de IteadStudio. Con él se puede manejar hasta 2 motores DC (con un consumo de hasta 2 A cada uno) o un motor paso a paso. Como los anteriores shields, es recomendable que se alimente mediante una fuente externa conectada a los bornes adecuados del mismo, la cual puede alimentar a su vez a la placa Arduino o no, según la colocación de un jumper específico. También puede ser alimentada si hay una fuente externa conectada directamente al zócalo "jack" de la placa Arduino. Como mayor novedad, este shield aporta un zócalo XBee para poder insertar un módulo de este tipo y así poder controlar el shield inalámbricamente. Esto está especialmente pensado para la construcción de vehículos teledirigidos. No utiliza ninguna librería propia para el manejo de los motores.

Otro shield que no necesita ninguna librería de terceros para ser utilizado es el shield "Ardumoto" de Sparkfun, el cual también está basado en el chip L298 (así que también es capaz de controlar 2 motores DC o un motor paso a paso). La alimentación de los motores se obtiene a través del pin "Vin" de la placa Arduino, por lo que es allí donde deberíamos conectar la fuente de alimentación externa. Fuentes adecuadas para este shield son por ejemplo el producto nº 298 de Sparkfun (un adaptador AC/DC de 9 V) , o bien los productos nº 9703 y 9704 (baterías LiPo). Este shield también incorpora un regulador interno que, a partir del voltaje de "Vin", ofrece una tensión de 5 V para proteger su circuitería más delicada.

El shield "Ardumoto" dispone de los bornes "OUT1" y "OUT2" para conectar los terminales de un motor DC, y los bornes "OUT3" y "OUT4" para conectar los del otro. A partir de aquí, para especificar el sentido de giro del primer motor deberemos utilizar en nuestros sketches el pin digital nº 12, ya que este es el pin conectado internamente a la patilla correspondiente del L298; según se envíe por ese pin un valor HIGH o LOW el motor girará en un sentido o en otro. Observar que tan solo se requiere una única conexión para especificar el sentido de giro (en vez de dos como hemos visto en hardware anterior). Igualmente, para especificar la velocidad de giro de ese primer motor deberemos utilizar en nuestros sketches el pin PWM nº 3. Si trabajamos con un segundo motor el sentido de giro se manipula mediante el pin digital nº 13 y la velocidad de giro mediante el pin PWM nº 11.

Ejemplo 6.30: Como muestra, he aquí este código, el cual maneja un solo motor.

```
/*En esta placa, el motor "A" es el conectado a los terminales 1 y 2,
y el motor "B" es el conectado a los terminales 3 y 4*/
int pwm_a = 3;  //Pin de control de velocidad del motor A
```



```
int dir_a = 12; //Pin de control de sentido de giro del motor
int i = 0;
void setup(){
  pinMode(pwm_a, OUTPUT);
  pinMode(dir_a, OUTPUT);
  analogWrite(pwm_a, 100);
}
void loop(){
/*La siguiente orden equivale a establecer la patilla n° 1 del chip
para el control de sentido de giro a HIGH y la patilla n° 2 a LOW,
pero no gira el motor hasta que no se le aplique una señal PWM */
  digitalWrite(dir_a, HIGH);
  fadein();   delay(1000);
  fadeout();  delay(1000);
//Paro el motor
  analogWrite(pwm_a, 0);   delay(2000);
/*La siguiente orden equivale a establecer la patilla n° 1 del chip
para el control de sentido de giro a LOW y la patilla n° 2 a HIGH,
pero no gira el motor hasta que no se le aplique una señal PWM */
  digitalWrite(dir_a, LOW);
  fadein();   delay(1000);
  fadeout();  delay(1000);
//Paro el motor
  analogWrite(pwm_a, 0);   delay(2000);
}
void fadein(){
  for(i = 0 ; i <= 255; i+=5) {
    analogWrite(pwm_a, i);
    delay(30);  //Me espero para ver el resultado
  }
}
void fadeout(){
  for(i = 255 ; i >= 0; i -=5) {
    analogWrite(pwm_a, i); delay(30);
  }
}
```

Sparkfun distribuye otro shield llamado “Monster Motor Shield”, que es una versión modificada del “Ardumoto” donde se ha sustituido el chip L298 por un par de chips VNH2SP30 y se ha reforzado la circuitería interna para soportar voltajes más elevados.

Otro shield que incorpora el chip L298P es el "2A Motor shield" de DFRobot. Como su nombre indica, puede ofrecer hasta un máximo de 2 A de corriente a los dos motores DC que se le conecten. Puede ser alimentado a través de la fuente conectada al conector "jack" de la placa Arduino (funcionando entonces a un voltaje entre 7 y 12 V), o bien de forma independiente de una fuente externa (funcionando entonces a un voltaje entre 5 y 35 V); para cambiar de un tipo a otro de alimentación se ha de modificar la ubicación de un jumper determinado del shield. El control del motor DC conectado a los terminales "M1+" y "M1-" se efectúa mediante el pin digital nº 4 (para especificar el sentido de giro) y el pin PWM nº 5 (para especificar la velocidad de giro). El control del motor DC conectado a los terminales "M2+" y "M2-" se efectúa mediante el pin digital nº 7 y el pin PWM nº 6. DFRobot también distribuye otro shield llamado "1A Motor Shield" que viene con el chip L293, por lo que solo permite un máximo de 1 A de corriente para ser consumida por los motores conectados.

Otro shield muy similar a los anteriores es el "Motor shield" de Open-Electronics, el cual incorpora el chip L298P, por lo que también puede manejar tanto motores DC como paso a paso con una corriente máxima de salida de 2 A por cada canal de salida. La alimentación puede ser externa o proveerla la propia placa Arduino.

En otra escala trabajan los shields Megamoto y Megamoto Plus de Robot Power, los cuales permiten controlar un motor DC en ambos sentidos o bien dos motores DC en el mismo sentido de giro con un consumo máximo de hasta 40 A y 25 V (es decir, hasta ¡1100W!).

Un shield específico para la conexión y uso de hasta 16 servomotores con solo la utilización de 4 pines digitales (del nº 6 al 9) es el "Renbotics ServoShield" de Seeedstudio. Además, dispone de una amplia zona para realizar prototipados. Se ha de programar con una librería propia, disponible en la página web del producto.

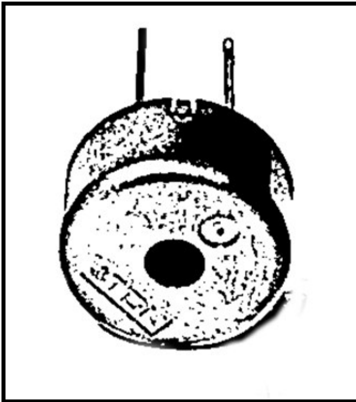
Un shield específico para la conexión y uso de hasta 2 motores paso a paso es el "Arduino dual step motor driver shield" de IteadStudio. Funciona con una alimentación de entre 5 V y 30 V y no requiere ninguna librería específica para ser programada. Se basa en el chip controlador de steppers A3967.

Un módulo –que no shield– especializado en el control de motores steppers muy parecido al shield anterior de IteadStudio (ya que está basado en el mismo chip A3967) es el llamado "Easy Driver" (comercializado entre otros por Sparkfun con código de producto 10267). Recomiendo la lectura del sencillo tutorial disponible en

su web (http://www.schmalzhaus.com) para comprender de una manera rápida su funcionamiento y sus capacidades.

EMISIÓN DE SONIDO

Uso de zumbadores



Un zumbador piezoeléctrico (en inglés, “buzzer” o “piezobuzzer”) es un dispositivo que consta internamente de un disco de metal, que se deforma (debido a un fenómeno llamado piezoelectricidad) cuando se le aplica corriente eléctrica. Lo interesante es que si a este disco se le aplica una secuencia de pulsos eléctricos de una frecuencia suficientemente alta, el zumbador se deformará y volverá a recuperar su forma tan rápido que vibrará, y esa vibración generará una onda de sonido audible.

Recordemos que un sonido aparece cuando vibra alguna fuente y estas vibraciones son transmitidas en forma de onda a través de algún medio elástico (como el aire o el agua). Si la frecuencia de esa onda está dentro de un rango determinado (el llamado “espectro audible”, el cual va de 20 Hz a 20 KHz), cuando llega al oído humano esas oscilaciones en la presión del aire son convertidas en el sonido que nuestro cerebro percibe.

Cuanto mayor sea la frecuencia de la onda sonora, más agudo será el sonido resultante (y al revés: cuanto menor es esa frecuencia, más grave es el sonido). Por tanto, para generar diferentes tonos con nuestro zumbador deberemos hacer vibrar la membrana a distintas frecuencias. Para ello, deberemos emitir desde nuestro Arduino pulsos digitales con la frecuencia deseada. Los zumbadores admiten voltajes desde 3 a 30 voltios, así que los 5 V de una salida digital de Arduino los soporta perfectamente.

Así pues, para hacer sonar un zumbador típico (como por ejemplo, el producto nº 160 de Adafruit o similar), simplemente deberemos conectar uno de sus terminales a tierra y el otro a un pin-hembra digital del Arduino configurado como salida digital. Atención, dependiendo del modelo de zumbador, este puede ser un componente polarizado o no, por lo que hay que fijarse si tiene alguna marca indicando la polaridad de sus terminales. Si se desea, al zumbador se le puede

conectar en serie un divisor de tensión (de 100 Ω está bien): pero hay que tener en cuenta que cuanta mayor resistencia tenga este divisor, menor será el volumen del sonido generado.

Ejemplo 6.31: Una vez realizadas las conexiones, podemos ejecutar este sketch:

```
void setup (){
     pinMode (8, OUTPUT); //Pin al que está conectado el zumbador
}
void loop (){
     digitalWrite (8, HIGH);
     delayMicroseconds (1000);
     digitalWrite (8, LOW);
     delayMicroseconds (1000);
}
```

Lo que estamos haciendo es enviar consecutivamente pulsos digitales HIGH y LOW a una velocidad tan elevada que hace vibrar el zumbador a una frecuencia audible por el ser humano. Concretamente, el pulso HIGH lo hacemos durar 1 ms y el pulso LOW otro 1 ms, por lo que el período de esta onda cuadrada es de 2 ms (0,002s). Por tanto, como f = 1/T, la frecuencia de la señal audible generada será de 1/0,002 = 500 Hz.

Ejemplo 6.32: Podemos hacer que la frecuencia del sonido vaya cambiando. Por ejemplo, el siguiente sketch reproduce (una sola vez) un sonido que pasa de ser grave a ser agudo de forma continua. Esto es porque al principio su periodo es de 10000 ms (es decir, 1/0,01 = 100 Hz) y al final su periodo es de 100 ms (es decir, 1/0,0001 = 10 KHz). Fijarse que podemos variar la duración del sonido si modificamos el incremento (es decir, el tercer parámetro del for, que en este caso es un decremento). Para probar este código, se ha de usar el mismo circuito que el ejemplo anterior.

```
int tono = 10000; //Nota inicial. Es el período de la onda
void setup (){
     pinMode (8, OUTPUT);
     for (tono=10000; tono>=100; tono=tono-10){
          digitalWrite (8, HIGH);
          delayMicroseconds (tono/2);
          digitalWrite (8, LOW);
          delayMicroseconds (tono/2);
     }
}
void loop (){}
```


Al ejecutar el código anterior notarás que el zumbador permanece mucho rato emitiendo sonidos graves (parece que le cuesta "avanzar") para finalmente "acelerar" y emitir los sonidos más agudos en muy poco tiempo. Podemos mitigar este efecto, y hacer que el sonido transcurra desde los sonidos graves hasta los sonidos agudos de una forma más equilibrada realizando dos cambios en el código anterior. El primer cambio es convertir la variable "tono" en una variable de tipo "float". El segundo cambio es sustituir el decremento actual del bucle "for" (es decir, `tono=tono-10`) por otro tipo de decremento; concretamente, cambiar la resta por una división. Si hacemos esto (es decir, si escribimos por ejemplo `tono=tono*0.97`) controlaremos mejor la "velocidad" de la sirena. Podemos cambiar este factor 0,97 por otro, como 0,99 o 0,93 para observar qué ocurre.

Ejemplo 6.33: Añadamos ahora al circuito anterior un potenciómetro. La patilla de un extremo (cualquiera) la conectaremos a la misma tierra que el zumbador, la patilla del otro extremo a la alimentación (que puede ser proporcionada por la propia placa Arduino a través del pin "5V") y la patilla central a un pin de entrada analógica de Arduino (por ejemplo, el nº 0). Gracias al sketch siguiente, seremos capaces de cambiar el tono del sonido generado por el zumbador simplemente girando la rueda del potenciómetro.

```
int tono = 1000; //Nota inicial. Es el período de la onda
void setup (){
     pinMode (8, OUTPUT);
}
void loop (){
     digitalWrite (8, HIGH);
     delayMicroseconds (tono/2);
     digitalWrite (8, LOW);
     delayMicroseconds (tono/2);
     tono=analogRead(0);
     tono=map(tono,0,1023,1000,5000);
}
```

Lo que hacemos en el código anterior es obtener la lectura analógica del potenciómetro (que va de 0 a 1023) y utilizarla para definir el tiempo que duran los pulsos HIGH y LOW enviados al zumbador. Cuanto menos duren, más agudo será el sonido. Fijémonos que el valor leído primero se mapea para que esté dentro de un rango entre 1000 y 5000 (aunque perfectamente se puede elegir otro rango) y que la duración de los pulsos es "tono/2" porque el valor de "tono" representa el período total de la onda cuadrada (es decir, un pulso HIGH más un pulso LOW). Si hacemos los

cálculos de párrafos anteriores, veremos que con el período de la onda cuadrada establecido entre 1000 y 5000, este sketch puede emitir sonidos entre 1 KHz y 200 Hz.

Si queremos variar el volumen del sonido emitido además de su frecuencia, deberemos de sustituir el divisor de tensión conectado en serie al zumbador por un segundo potenciómetro. Concretamente, deberíamos conectar un extremo de ese nuevo potenciómetro al pin-hembra digital del Arduino configurado como salida por donde se emite el sonido, el otro extremo del potenciómetro a tierra y su patilla central a un terminal del zumbador. De esta manera, podremos regular la intensidad de corriente recibida por el zumbador y, por tanto, el volumen del sonido emitido.

Otra manera de regular el volumen podría ser conectando el zumbador a una salida analógica de la placa en vez de a una digital. No obstante, de esta manera no podemos alterar la frecuencia de la onda emitida (ya que en un pin PWM esta es constante a 490 Hz), por lo que el sonido que surge no es especialmente agradable.

Las funciones tone() y noTone()

Afortunadamente, normalmente no tendremos que utilizar *digitalWrite()* ni *analogWrite()* para emitir pitidos porque el lenguaje Arduino ofrece dos funciones especialmente pensadas para ello que nos facilitan mucho la escritura de nuestros sketches.

tone(): genera una onda cuadrada de una frecuencia determinada (especificada en Hz como segundo parámetro –de tipo “word”–) y la envía por el pin digital de salida especificado como primer parámetro, el cual deberá estar conectado algún tipo de zumbador o altavoz. Esta función no es bloqueante; esto quiere decir que una vez comienza a emitirse el sonido, el sketch sigue su ejecución en la siguiente línea de código. La duración de la onda (en milisegundos) se puede especificar opcionalmente como tercer parámetro –su tipo es “unsigned long”– ; si no se indica, la onda se emitirá hasta que se llame a la función *noTone().* La forma de la onda cuadrada no se puede alterar: siempre tiene un valor alto la mitad de su período y un valor bajo la otra mitad (lo que se llama tener un ciclo de trabajo del 50%). Tampoco puede alterar el volumen del sonido emitido. En la documentación oficial se advierte que el uso de esta función puede desestabilizar la salida PWM de los pines 3 y 11 en la placa Arduino UNO. Esta función no retorna nada.

Solo una señal de audio puede generarse en un momento determinado: si ya hubiera una señal emitiéndose por otro pin diferente del especificado en *tone()* (porque tenemos más de un zumbador en nuestro circuito), la señal anterior continuaría emitiéndose como si nada y la nueva no se llegaría a emitir. Si esa señal previa estuviera emitiéndose en el mismo pin, entonces el efecto sí que sería la sustitución de la señal anterior por la nueva. Por lo tanto, si se desea emitir diferentes señales en múltiples pines, se deberá parar la emisión en uno (con *noTone()*) para empezar la emisión en otro.

noTone(): deja de generar la onda cuadrada que en principio estaba emitiéndose (por una ejecución previa de *tone()*) a través del pin especificado como (único) parámetro. Si no hay ninguna señal emitiéndose en ese pin, esta función no hace nada. No tiene valor de retorno.

Ejemplo 6.34: Un sketch muy sencillo que muestra cómo trabajan estas funciones es el siguiente, y que se puede probar utilizando un circuito con solo un zumbador conectado a tierra y al pin digital de salida nº 8 de la placa Arduino (opcionalmente a través de un divisor de tensión). Al ejecutarlo se podrá escuchar un sonido ininterrumpido de sirena.

```
int duracion = 250;  //Duración del sonido
int freqmin = 2000;  //La frecuencia más baja a emitir
int freqmax = 4000;  //La frecuencia más alta a emitir
void setup(){
	pinMode(8, OUTPUT);
}
void loop(){
	int i;
	//Se incrementa el tono (se hace más agudo)
	for (i = freqmin; i<=freqmax; i++){
		tone (8, i, duracion);
	}
	//Se disminuye el tono (se hace más grave)
	for (i = freqmax; i>=freqmin; i--){
		tone (8, i, duracion);
	}
}
```

Ejemplo 6.35: Otro código más sofisticado es el siguiente (con el mismo circuito), en el cual se reproduce una melodía. Para lograrlo, el sketch emite, una tras otra y durante el tiempo preciso, las frecuencias exactas correspondientes a las notas musicales de esa melodía.

```
//Frecuencias de las notas de la melodía
int melodia[] = {262, 196, 196, 220, 196, 247, 262};
/*Duración de las notas
(4 = dura un cuarto de tiempo, 8=dura una octava parte, etc.)*/
int duracionNota[] = {4,8,8,4,4,4,4,4 };
void setup() {
	int i;
	//Recorro las 7 notas de la melodía una tras otra
	for (i = 0; i < 8; i++){
/*Para calcular la duración de la nota, se divide un segundo por la
cantidad marcada en duracionNota[]. Por ejemplo, un cuarto de tiempo
son 1000/4 segundos, una octava parte son 1000/8 segundos, etc. */
		tone(8, melodia[i], 1000/duracionNota[i]);
/*Como la función tone() no es bloqueante, el sketch sigue
ejecutándose sin parar después de ella. Para evitar volver arriba del
loop enseguida y distinguir las notas, establezco un tiempo mínimo
entre ellas (la duración de la nota + 30% parece ir bien) parando el
sketch */
		delay(1300/duracionNota[i]);
		// Se deja de emitir la nota
		noTone(8);
	}
}
void loop(){}
```

En el sketch anterior hemos usado una serie de frecuencias concretas. ¿A qué notas musicales corresponden? ¿Cómo hemos sabido su valor? Bueno, lo primero que hemos de saber es que todo el conjunto de notas se dividen en “paquetes” que se llaman “octavas”, y cada octava tiene doce notas: do, do# (el símbolo “#” se lee “sostenido”), re, re#, mi, fa, fa#, sol, sol#, la, la# y si. Pero dentro del espectro audible hay una decena de octavas, por lo que en realidad hay varios dos, varios res, etc. Para distinguir las notas con el mismo nombre de diferentes octavas, tras su nombre se les añade un número indicando a qué octava pertenece; así podemos tener la nota mi4, la nota sol3, etc. Lo importante de todo esto es saber que una nota de una octava concreta (pongamos que el fa2) se corresponde con una onda de exactamente la mitad de frecuencia que la nota con el mismo nombre de la octava superior (en este caso, fa3).

Existe una fórmula matemática que permite obtener las frecuencias de todas las notas de todas las octavas. Dada la nota que queremos (la letra “n” de la fórmula, que ha de ser un entero entre 1 y 12: d o= 1, do# = 2,re=3….hasta si = 12) y dada la

octava donde está (la letra "o" de la fórmula, que ha de ser un entero entre 1 y 10), se puede saber su frecuencia si calculamos la fórmula $440 \cdot e^{((o-3)+\frac{n-10}{12})\cdot \ln(2)}$.

Ejemplo 6.36: El siguiente sketch muestra las frecuencias correspondientes a 10 octavas; cada octava se mostrará por el "Serial monitor" separada por una línea de guiones. Las octavas que solemos escuchar en la mayoría de canciones son la 2ª, 3ª y 4ª.

```
void setup(){
   int i,j;
   Serial.begin(9600);
   for(i=1; i<=10; i++){  //Recorro las escalas
       Serial.print("-------------Escala ");
       Serial.println(i);
       for(j=1; j<=12; j++){ //Recorro las notas de esa escala
        //j=1 es un do, j=2 es un do#, j=3 es un re...
             Serial.print(j);
             Serial.print("   ");
             Serial.println(frecuencia(i,j));
       }
   }
}
void loop(){}
float frecuencia(float octava, float nota) {
   return (440.0*exp(((octava-3)+(nota-10)/12)*log(2)));
}
```

Ejemplo 6.37: Otro código algo más sofisticado que también reproduce una melodía, es el siguiente. Para ejecutarlo necesitamos el mismo circuito del ejemplo anterior:

```
int longitud = 15; // Número de notas de la canción
/*El array notas[] tiene las notas de la canción
Se emplea la notación americana:
do=c, re=d, mi=e, fa=f, sol=g, la=a, si=b
Un espacio representa un silencio */
char notas[] = "ccggaagffeeddc ";
/*Duración de cada nota de la canción.
"2" dura el doble que "1", "4" dura el doble que "2", etc*/
int pulsacion[] = {1,1,1,1,1,1,2,1,1,1,1,1,1,2,4};
//Nombres de las notas de una escala
char nombres[] = { 'c', 'd', 'e', 'f', 'g', 'a', 'b', 'C' };
//Sus correspondientes frecuencias
int tonos[] = { 523, 587, 659, 698, 783, 880,98, 1047 };
```

```
int tempo = 350;
void setup() {
  pinMode(8, OUTPUT);
}
void loop() {
  int i;
  for (i = 0; i < longitud; i++) {
      if (notas[i] == ' ') {
          //Si hay silencio, se espera
          delay(pulsacion[i] * tempo);
      } else {
          //Si no, emite la nota
          playNota(notas[i]);
          //con la duración especificadas
          delay(pulsacion[i]*tempo);
          noTone(8);
      }
  }
}
void playNota(char nota) {
  int i;
  /*Emite la frecuencia correspondiente al nombre de la nota. Para
  ello, busco en el array nombres[] si la nota especificada como
  parámetro está allí.  */
  for (i = 0; i < 8; i++) {
      if (nombres[i] == nota) {
          tone(8,tonos[i]);
      }
  }
}
```

Ejemplo 6.38: Y otro código de ejemplo más, muy parecido al anterior. En este caso, el mismo circuito reproducirá la nota musical especificada a través del canal serie.

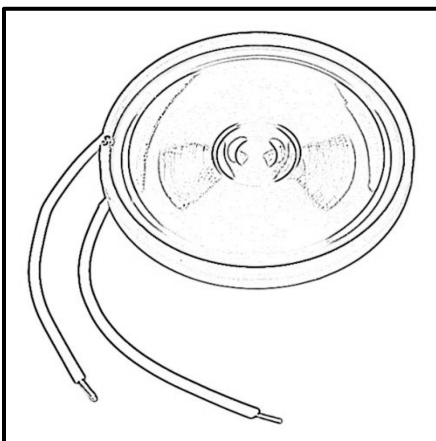
```
char nombres[] ={'c', 'd', 'e', 'f', 'g', 'a', 'b', 'C'};
int tonos[] = { 523, 587, 659, 698, 783, 880, 983, 1047 };
char caracter;
void setup() {
  pinMode(8, OUTPUT);
  Serial.begin(9600);
}
void loop() {
  caracter = Serial.read();
  if (caracter != -1) {    //Si se ha escrito algo en el canal serie…
```



```
       playNota(caracter); //…intento emitir la nota correspondiente
   }
}
void playNota(char nota) {
  int i;
  for (i = 0; i < 8; i++) {
  /*Si el carácter escrito en el "Serial monitor" coincide con alguno
  correspondiente a alguna nota dentro del array, se reproduce. Si
  no, hay silencio */
      if (nombres[i] == nota) {
          tone(8,tonos[i]);
          delay(500);  //La emisión de la nota dura medio segundo
          noTone(8);
      }
  }
}
```

Las funciones *tone()* y *noTone()* incluidas por defecto dentro del lenguaje Arduino en realidad son versiones simplificadas de funciones similares pertenecientes a una librería llamada "Tone", descargable desde http://code.google.com/p/rogue-code. Entre las ventajas que ofrece la librería "Tone", está la de poder reproducir varios tonos simultáneamente (lo que se llama "polifonía") o la de tener ya predefinidas un conjunto de constantes correspondientes a las diferentes notas musicales.

Uso de altavoces



En vez de zumbadores, podemos utilizar altavoces sencillos de poca potencia (0,5 W, por ejemplo), como los productos 9151 o 10722 de Sparkfun. Al igual que los zumbadores, los altavoces generan ondas acústicas a partir de las señales eléctricas que reciben, pero el mecanismo físico para lograrlo es mucho más sofisticado, por lo que podremos lograr una calidad y un volumen de sonido bastante mayor.

Para conectar los altavoces a nuestro circuito, debemos enchufar uno de sus dos terminales a tierra y el otro a un pin-hembra digital del Arduino configurado como salida, que

será por donde enviaremos los pulsos de señal generados por la función *tone()*. Los altavoces suelen ser componentes polarizados, por lo que hay que fijarse si tiene alguna marca para indicar la polaridad de cada terminal (normalmente, el terminal positivo tiene un cable rojo y el negativo uno negro).

Es muy importante conectar un divisor de tensión en serie al altavoz para no dañar la placa. El valor mínimo de este divisor ha de ser de 100 Ω, aunque si se utiliza un valor mayor, también estará bien. Hay que tener en cuenta, no obstante, que cuanta mayor resistencia tenga el divisor, menor será el volumen de sonido que puede generar el altavoz. En este sentido, es recomendable utilizar un potenciómetro (preferiblemente logarítmico) como divisor de tensión variable para así poder controlar el volumen del sonido convenientemente.

En configuraciones más sofisticadas a veces también se conecta un condensador “by-pass” (de 0,1 µF está bien) entre los dos terminales del altavoz para protegerlo de posibles picos de señal provenientes del pin de salida digital de la placa cuando esta cambia su estado de HIGH a LOW, o viceversa.

Además, muchas veces el altavoz suele venir también acompañado de un condensador conectado en serie a él. La función de este condensador es actuar como filtro “pasa-altos”. Un filtro “pasa-altos” conduce muy bien las señales eléctricas correspondientes a las frecuencias de sonido altas (concretamente, a las que son mayores que una determinada frecuencia llamada “frecuencia de corte”, diferente según las características concretas de cada condensador) pero ofrece mucha resistencia a (y por tanto, elimina) las frecuencias de la señal que están por debajo de esa frecuencia de corte. Entre otras aplicaciones, los filtros pasa-altos se suelen utilizar mucho para hacer desaparecer la señal de frecuencia 0 (correspondiente a una señal eléctrica estable y continua que siempre aparece como “colchón” permanente bajo el resto de señales que forman realmente el sonido), ya que tan solo sirve para calentar los altavoces y malgastar energía inútilmente consumiendo más rápido la carga de las pilas/baterías (si es el caso).

En otro orden de cosas, es importante conocer las dos características técnicas más relevantes de un altavoz: su resistencia y su potencia de trabajo. La resistencia del altavoz se suele llamar “impedancia” (para señalar que, en realidad, esta característica no tiene un valor constante sino que depende de la frecuencia de la señal, pero en esto no profundizaremos). En la mayoría de proyectos donde interviene una placa Arduino los altavoces más habitualmente utilizados son los de 4 Ω o 8 Ω nominales.

La potencia de trabajo de un altavoz es la cantidad de energía que puede consumir en un segundo sin problemas (recordemos que está definida como $P = V \cdot I = I^2 \cdot R = V^2/R$), más allá de la cual podría dañarse. Este dato nos sirve para conocer la "cantidad de volumen" máximo de sonido que puede proporcionar ese altavoz, ya que cuanto mayor sea su potencia de trabajo, mayor volumen será capaz de emitir.

Si conectáramos directamente (sin divisor de tensión) un altavoz de 8 Ω a un pin de salida de nuestra placa Arduino (el cual ya sabemos que aporta 5 V en estado HIGH), la potencia consumida (y por tanto, el volumen emitido) sería la máxima posible: $P = (5V)^2/8\Omega = 3{,}125$ W. No obstante, suponiendo que el altavoz soporte esta potencia, tendríamos un problema: la intensidad que fluiría por el pin de salida sería $I = V/R = 5V/8\Omega = 625$ mA (también podríamos haber llegado al mismo resultado mediante: $I = (P/R)^{1/2} = (3{,}125W/8\Omega)^{1/2} = 625$ mA) , cantidad que es muy superior al límite máximo de intensidad soportado por estos pines, el cual ronda los 40/50 mA. Por tanto, para no freír estos pines es necesario incluir un divisor de tensión que reduzca la intensidad y potencia generadas a los niveles aceptados por la placa. De ahí lo comentado en párrafos anteriores sobre la necesidad de conectar una resistencia en serie al altavoz.

Sin embargo, tampoco es bueno pasarse en el valor del divisor de tensión: si conectáramos por ejemplo un resistor de 4,7 KΩ entre un pin de salida de la placa Arduino y ese mismo altavoz, la tensión a la que estaría sometido dicho altavoz sería $V = 8\Omega \cdot 5V/(8\Omega + 4{,}7K\Omega) = 8{,}5$ mV, y por tanto tan solo recibiría una intensidad de $I = V/R = 8{,}5mV/8\Omega \approx 1$ mA. Esto implicaría que la potencia recibida sería de 0,008 milivatios ($P = V \cdot I = 0{,}0085\ V \cdot 0{,}001ª = 0{,}0000085$ W). Es decir, un sonido inaudible. Si probamos en cambio con un valor para el resistor de por ejemplo 120 Ω, la tensión a la que se sometería entonces el altavoz sería de $V = 8\Omega \cdot 5V/(8\Omega + 120\Omega) = 312{,}5$ mV y la intensidad recibida sería de $I = V/R = 312{,}5mV/8\Omega \approx 39$ mA, con lo que la potencia resultante sería de $P = V \cdot I = 0{,}3125\ V \cdot 0{,}039ª = 12.2$ mW. En este caso vemos que aun aportando por el pin de salida el máximo de corriente admitida, la potencia generada (aunque mucho mayor que en el caso del divisor de 4,7 KΩ) sigue siendo no demasiado elevada. Esto nos lleva a una conclusión: si deseamos más volumen, deberemos utilizar un amplificador.

Amplificación simple del sonido

Un amplificador sirve para aportar a un altavoz una determinada potencia, mayor de la que aporta la fuente del sonido (en nuestro caso, la placa Arduino) por sí misma. A más potencia recibida, el altavoz podrá vibrar con más fuerza y emitir el

sonido a un volumen mayor. Para conseguir su objetivo, los amplificadores pueden “jugar” con aumentar o bien el voltaje, o bien la intensidad aportados al altavoz, o bien una combinación de ambos (esto es fácil verlo si recordamos que P = V·I).

Lo más sencillo en la práctica es utilizar amplificadores en forma de placas breakout, ya que nos facilitan las conexiones y además incorporan circuitería suplementaria que estabiliza las señales. La idea es, en general, por un lado conectar el pin digital de salida de audio de la placa Arduino a la entrada adecuada de la plaquita-amplificador, por otro lado conectar la salida de la plaquita-amplificador a la entrada de señal del altavoz, y compartir una tierra común con los tres dispositivos implicados: Arduino, plaquita-amplificador y altavoz.

Una placa breakout que incluye un amplificador de audio (concretamente, el TPA2005D1) es el producto nº 11044 de Sparkfun. Esta placa, alimentada con 5 V puede proporcionar a un altavoz de 8 Ω una potencia de hasta 1,4 W con una distorsión máxima (lo que técnicamente se llama “THD + N” (es decir, Total Harmonic Distorsion + Noise) del 0,2%. Tiene un conector etiquetado como “IN+”, donde deberemos enchufar la salida digital de la placa Arduino por donde emite *tone()* y un conector “IN-“, que ha de ir a tierra; por otro lado tiene dos conectores más (“OUT+” y “OUT-“) para el altavoz de 8 Ω. Finalmente, tiene un conector para su alimentación, otro para tierra y otro etiquetado como SDN, el cual, si se conecta a tierra o a una señal LOW apagará el amplificador (para ahorrar consumo). También admite la posibilidad de soldar un potenciómetro de 10 KΩ (aunque no lo incorpora de serie), para ajustar el nivel de amplificación del sonido.

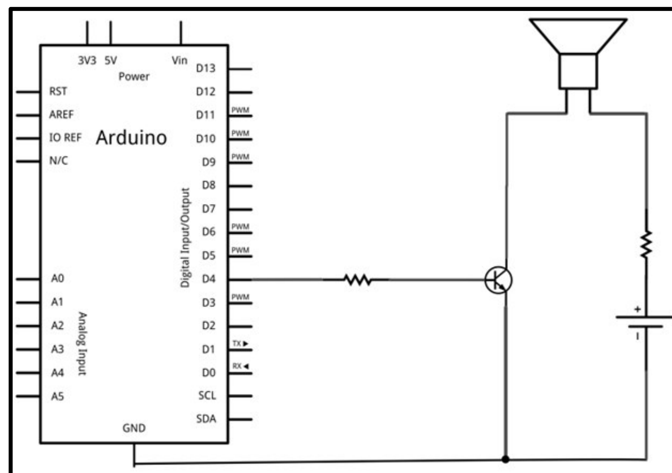
Otra placa muy parecida es la llamada “LM4889 Audio amplifier”, distribuida por Modern Devices. Esta placa, alimentada con 5 V puede proporcionar a un altavoz de 8 Ω una potencia de hasta 1 W con una distorsión máxima del 0,2%. Esta placa tiene un solo conector para la entrada de sonido proveniente de la placa Arduino, dos conectores (+ y -) para el altavoz de 8 Ω, un conector para la alimentación, otro para tierra y otro etiquetado como SHD, el cual, si se conecta a tierra o a una señal LOW apaga el amplificador (para ahorrar consumo). También dispone de un potenciómetro ya preensamblado para poder alterar la potencia proporcionada y así regular el volumen de salida.

De todas formas, podemos aumentar el volumen del sonido emitido por nuestra placa Arduino sin necesidad de utilizar ningún amplificador específico como los descritos en párrafos anteriores. Estos dispositivos están especializados en reproducir ondas acústicas complejas sin alterar el sonido original y mantener la proporción de frecuencias intacta, pero en realidad, la placa Arduino no es capaz de

"generar" audio real: la función *tone()* tan solo emite una onda cuadrada que oscila entre 5 V y 0 V a una determinada frecuencia única. Esta gran simplicidad de la onda permite que para amplificar esta señal (es decir, para aumentar la potencia enviada al altavoz) nos baste con un simple transistor bipolar y nada más.

La idea es simplemente aumentar la intensidad de corriente que circula por el altavoz sin aumentar la que sale del pin de la placa Arduino por donde se emite la señal a escuchar (la cual está limitada, recordemos, a tan solo 40/50 mA). En el esquema siguiente se muestra un circuito donde se implementa esta idea. Tal como se puede ver, se utiliza un transistor NPN (como por ejemplo el 2N4401 u otro similar) como amplificador. Su funcionamiento es muy parecido al que vimos cuando estudiamos el control de motores DC: la única diferencia destacable es que ahora la base del transistor está conectada a un pin de salida digital (en vez de a una salida PWM), porque allí será por donde se emitirán los tonos musicales mediante el uso de *tone()*, función cuyo comportamiento, recordemos, es totalmente digital. La resistencia conectada a la base del transistor con ser de 100 Ω ya es suficiente.

La fuente de alimentación externa puede ser de 5 V o más, siempre que la potencia aportada al altavoz no exceda la máxima admitida. Es fácil calcular, usando la Ley de Ohm, cuánta potencia aporta si conocemos la tensión generada por la fuente y el valor del divisor de tensión del altavoz (es decir, de la resistencia conectada en serie a él). De todas formas, aunque se elija una fuente de 5 V, es muy recomendable no utilizar la propia placa Arduino como fuente para evitar posibles ruidos e inestabilidades en la señal.



Sonidos pregrabados

Añadir audio de calidad a un proyecto con Arduino es difícil: el microcontrolador que incorpora no es lo suficientemente capaz para manejar la cantidad de datos necesaria para poder reproducir sonidos de calidad. No obstante, existen algunos proyectos que intentan superar estas limitaciones.

La librería "SimpleSDAudio"

Uno de ellos es la librería "SimpleSDAudio", descargable desde http://hackerspace-ffm.de/wiki/index.php?title=SimpleSDAudio, que permite reproducir ficheros de sonidos almacenados en una tarjeta SD con una calidad decente con un mínimo de hardware adicional. Concretamente, esta librería es capaz de reproducir audio con una resolución de 8 bits y una frecuencia de muestreo de 62, 5KHz ("fullrate") o bien de 31,25 KHz ("halfrate"), en modo mono o modo estéreo. Eso sí, requiere 1,3 kilobytes de RAM para poder funcionar.

Breve nota sobre las características de un fichero de audio:

No todos los ficheros de audio son iguales: unos reproducen el sonido con más calidad que otros. Pero ¿qué significa exactamente que un sonido tenga "más calidad"? Cuando se captura un sonido (por medio de un micrófono o similar) se produce un proceso de conversión de la onda sonora (analógica) a pulsos eléctricos digitales (bits). La frecuencia a la que se toman muestras de esa onda sonora para guardarlas en un fichero es la llamada "frecuencia de muestreo". Es decir, este parámetro indica el número de veces por segundo que el micrófono ha tomado información del sonido. Es evidente que cuanta mayor sea la frecuencia de muestreo, la captura de la señal será más fidedigna al sonido real porque dejará menos intervalos sin medir. No confundir frecuencia de muestreo con la frecuencia del sonido. Los valores más usuales empleados en las grabaciones digitales son 11,025Hz para grabaciones de voz, 22,050Hz para grabaciones de música con calidad mediana y 44,100 Hz para grabaciones de música con alta calidad.

Otro dato importante es la resolución. Representa la cantidad de información que se captura en cada muestra del sonido. De nada sirve tener una gran frecuencia de muestreo si luego en cada muestra no se ha capturado cómo era el sonido en ese momento de la forma más precisa posible. Los valores más comunes para la resolución son 8 bits por lectura (por lo que solo se permiten 256 valores posibles diferentes para almacenar el estado del sonido en ese momento) y 16 bits por lectura (por lo que ofrece una escala de 65,536 y por lo tanto, permite capturas mucho más completas y precisas).

Además, tenemos otro dato a considerar, que es el número de canales: si se realiza un solo registro sonoro estamos hablando de una señal monofónica y se realizan dos registros simultáneos es una señal estereofónica.

Por otro lado, una vez grabado el audio en un fichero, podemos tener un problema: que ese fichero ocupe mucho espacio en disco. Es fácil comprobar que si multiplicamos el número de muestras por segundo (la "frecuencia de muestreo") por el número de segundos que dura la grabación por los bits almacenados en cada muestra por el número de canales usados, obtenemos una cantidad de bits muy elevada. Por tanto, en la mayoría de ocasiones, estos ficheros se guardan de forma comprimida.

Los códecs de audio son algoritmos matemáticos que permiten comprimir los datos, haciendo que ocupen mucho menos espacio. Hay códecs que comprimen más o menos que otros, a cambio de perder más o menos calidad de sonido. El problema es que el aparato reproductor de ese audio ha de poder entender el códec particular con el que se comprimió un determinado fichero, y si no es así, no lo podrá reproducir. Los códecs de audio más usuales son el conocido popularmente como "MP3" (blindado por un complejo sistema de patentes) o el "Vorbis" (basado en estándares abiertos y libres), entre muchísimos más. Si el fichero no está comprimido por ningún códec, se suele decir que está en formato PCM ("Pulse Code Modulation").

Normalmente, la extensión de un fichero de audio indica el códec que se ha utilizado en él (.mp3 para un fichero codificado en MP3, etc.), pero hay casos en que no es tan fácil. Por ejemplo, la extensión ".wav" es la que se acostumbra a emplear en Windows para identificar los ficheros de audio digital, pero los datos de los ficheros ".wav" pueden estar en formato PCM (sin comprimir) o pueden haber sido comprimidos con cualquiera de los códecs disponibles para Windows.

Los ficheros deben estar grabados en una tarjeta SD o SDHC porque la memoria EEPROM de la placa Arduino es muy limitada y es muy difícil poder guardar allí música de cierta duración. Deberemos, por tanto, utilizar un shield o módulo que incorpore un zócalo SD/microSD (o conectar directamente la tarjeta SD a nuestro circuito mediante una ristra de pines soldados a sus terminales).

La librería "SimpleSDAudio" asume que la tarjeta SD se comunica vía SPI con la placa Arduino, por lo que por defecto utiliza para ello los pines estándares 11, 12 y 13, y el nº 4 como canal CS. Si la tarjeta que tengamos conectada utilizara otro pin como canal CS, este se debería especificar entonces en el código fuente de nuestro

sketch. Además, requiere obligatoriamente el uso de los pines PWM 9 y 10 de la placa Arduino UNO para la salida del audio (si la salida es mono tan solo se requiere el pin 9). Por tanto, suponiendo que queremos conectar un altavoz mono a nuestra placa Arduino, la manera más sencilla sería enchufarlo por un lado al pin 9 a través de una resistencia en serie de 100 Ω y por otro a tierra. Otros ejemplos de conexión se muestran dentro del fichero “SimpleSDAudio.h”, incluido dentro de la librería.

El “Wave Shield” de Adafruit

La gente de Adafruit ha diseñado un shield (el “Wave Shield”) que permite reproducir ficheros de audio (de cualquier duración) que tengan las siguientes características: que sean “mono”, que tengan una frecuencia de muestreo de 22 KHz, una resolución de 12 bits y que no tengan compresión.

Los ficheros deberán estar grabados en una tarjeta SD (o SDHC) formateada en FAT16 o FAT32, la cual debemos tener introducida en el zócalo correspondiente del shield. Estos ficheros solo pueden estar almacenados en la carpeta “raíz”, y solo puede ser reproducido uno en cada momento.

El shield dispone de un zócalo “jack” de 3,5 mm estándar para poderle conectar unos auriculares, y también de dos conectores para enchufar los cables de un altavoz (los cuales se activan cuando detectan que no hay auriculares). Estos altavoces pueden ser de 8 ohmios o de 4 ohmios; en el primer caso recibirán una potencia de 1/8 W y en el segundo una de ¼ W gracias a un amplificador integrado en el shield, el TS922IN. En cualquier caso, el volumen de salida se puede controlar con un potenciómetro logarítmico también incorporado en el shield.

Los pines 10, 11, 12 y 13 del shield son usados por la tarjeta SD para comunicarla vía SPI con la placa Arduino. Los pines 2, 3, 4 y 5 del shield son usados por defecto por el conversor digital-analógico para comunicarse con la placa Arduino. Así pues, solo quedan disponibles los pines digitales 6, 7, 8 y 9, además de los pines de entrada analógicos (que pueden actuar también como pines de entrada/salida digitales extra).

Los ficheros han de estar en el formato mencionado anteriormente (22 KHz, 12 bits mono y sin comprimir) para ser reproducidos por el “Wave Shield” ya que el tratamiento de ficheros comprimidos requiere un chip especializado de potencia y precio elevado, y por eso no está incluido en este shield. Para saber si un fichero de audio tiene el formato requerido, podemos consultar la información mostrada en el cuadro que aparece en cualquier sistema operativo cuando hacemos clic con el botón

derecho sobre el icono de ese fichero y seleccionamos la opción "Propiedades". En caso de que el formato actual no sea el adecuado, una manera de convertirlo al que nos interesa es utilizando un editor de audio cualquiera, como por ejemplo Audacity (http://audacity.sourceforge.net), el cual es libre y multiplataforma.

Si usamos Audacity, el proceso es el siguiente: tras abrir el fichero, para convertir el fichero en mono (si no lo es ya) debemos seleccionar las dos pistas estéreo e ir al menú "Tracks > Stereo Track to Mono". Para pasarlo a una resolución de 12 bits (en realidad 16, pero también funciona) debemos clicar en el título de la pista y seleccionar del cuadro desplegable la opción llamada "Set Sample Format -> 16-bit". Para pasarlo a una frecuencia de 22 KHz (o menos) , debemos seleccionar del mismo cuadro desplegable anterior la opción "Set rate -> 20.050Hz". Para guardar el fichero en formato no comprimido, debemos usar la opción "Export as WAV" del menú "File" y elegir el formato "Other uncompressed files"; seguidamente hay que clicar en el botón "Options" y seleccionar el header "WAV (Microsoft)" y el enconding "Signed 16 bit PCM". Una vez hecho todo esto, podremos clicar finalmente en el botón "Save".

El "Wave Shield" controla mediante una librería específica desarrollada por la gente de Adafruit y descargable de http://code.google.com/p/wavehc. Esta librería nos permite reproducir sonidos cuando se pulsa un botón, cuando un sensor se activa, cuando un dato se recibe a través del canal serie, etc. El sonido se reproduce de forma asíncrona, por lo que la placa Arduino puede seguir trabajando mientras el sonido se está reproduciendo. Por razones de espacio no podemos profundizar en su estudio, pero afortunadamente está ampliamente documentada y proporciona muchos ejemplos. Concretamente, el sketch de ejemplo "dap_hc.pde" es el que permite reproducir todos los ficheros grabados en la tarjeta SD uno tras otro sin fin; recomiendo su estudio pormenorizado para comprender las interioridades de esta librería. Otros códigos de ejemplo interesantes y muy bien comentados se pueden encontrar en http://www.ladyada.net/make/waveshield/examples.html.

Shields que reproducen MP3

Si lo que deseamos es reproducir ficheros MP3 (a 192 Kbps) tal como lo solemos hacer en nuestro reproductor de música portátil o desde nuestro computador, existen shields adecuados para eso. Por ejemplo, el "MP3 Player Shield" de Sparkfun. Al igual que el "Wave Shield" de Adafruit, incluye un zócalo para insertar una tarjeta SD con los ficheros de audio, y ofrece una salida jack de 3,5 mm estéreo para los auriculares además de dos conexiones ("R/L" y "-") para enchufar un altavoz pequeño (o un amplificador externo). Este shield utiliza el chip decodificador de MP3

llamado VS1053B, el cual también es capaz de reproducir audio codificado en los códecs Vorbis, AAC y WMA, además de poder interpretar ficheros MIDI. Para trabajar con él en la página de Sparkfun proponen varios sketches de ejemplo, pero son de un nivel relativamente complicado; afortunadamente, existe una librería llamada "SFEMP3Shield" que facilita mucho su programación, descargable desde https://github.com/madsci1016/Sparkfun-MP3-Player-Shield-Arduino-Library .

Otro shield que es capaz de reproducir ficheros MP3 de algo más de calidad (con frecuencia de muestreo de 48 KHz y bitrate de 320 Kbps), así como también ficheros, PCM WAV es el "rMP3 Playback Module" de Rogue Robotics. Como salida de audio incorpora un conector jack estéreo de 3,5 mm (ideal para conectar unos auriculares, por ejemplo). Utiliza una librería propia llamada "Arduino-Library-RogueMP3" tanto para el acceso y manipulación de los ficheros ubicados en la tarjeta SD/SDHC como para el control de la reproducción. Esta librería se puede descargar de http://code.google.com/p/rogue-code.

Otro shield que puede reproducir ficheros MP3 (y Vorbis) es el "Music Shield" de Seeedstudio, también basado en el chip VS1053B. Se programa mediante una librería propia llamada "Music", descargable de la página del producto. Esta librería se basa en una librería independiente –pero incluida en la descarga– para acceder a los ficheros de la tarjeta SD, llamada "fat16lib" (http://code.google.com/p/fat16lib). La librería "fat16lib" es más rápida y ligera que la librería oficial de Arduino, pero solo puede trabajar con el formato FAT16, por lo que la tarjeta SD que se use con este shield solo puede estar formateada en FAT16. Como novedad, este shield incorpora un conector jack "LINE IN" que permite la grabación de sonido, aunque esta funcionalidad solamente vale cuando es acoplada a una placa Arduino Mega. Recomiendo consultar su documentación online, que es muy completa y clara.

Un shield algo diferente de los anteriores es el "Music Instrument Shield" de Sparkfun (producto nº 10587). Aunque incorpora el mismo chip VS1053B que el que viene en el "MP3 Player Shield", está preconfigurado para solo actuar como reproductor MIDI. Esto significa que mediante el envío de determinados comandos a través del canal serie, este chip es capaz de reproducir una gran cantidad de sonidos (pianos, vientos, percusiones, efectos especiales, etc.) que tiene pregrabados de fábrica. Incluso pueden sonar hasta 31 instrumentos diferentes a la vez. El shield ofrece un conector jack de 1/8'' para conectar un altavoz o unos auriculares. Recomiendo consultar la página http://www.sparkfun.com/tutorials/302 para comprender en profundidad el uso de este shield, además de leer y probar los muy ilustrativos códigos Arduino de ejemplo disponibles en la página del producto.

Módulos de audio

Si, en vez de shields, lo que queremos es utilizar módulos independientes capaces de almacenar ficheros de audio y de reproducirlos, tenemos unas cuantas alternativas. Por ejemplo, Sparkfun ofrece una placa breakout para el mismo chip VS1053B que viene en su “MP3 Player Shield”, a la cual se pueden conectar directamente las salidas de audio. En la página del producto (con código 9943) se ofrecen códigos Arduino de ejemplo de manejo de esta placa, pero son de un nivel más avanzado que el de este libro, así que recomendamos otras alternativas.

DFRobot fabrica la plaquita “DFRduino Player”. Tiene un zócalo para alojar una tarjeta SD formateada en FAT16 donde se han de almacenar los ficheros de audio y se basa en el chip decodificador de MP3 VS1003 (pudiendo reproducir ficheros en formato WAV, MP3 y MIDI). Se puede comunicar con nuestra placa Arduino por el canal serie o bien vía I^2C (seleccionable mediante un jumper). En el caso de utilizar la comunicación I^2C, además del cable de alimentación (5V) y tierra, se ha de usar un cable conectando el pin “DO” del módulo y el pin SDA de la placa Arduino, y otro conectando el pin “DI” del módulo y el pin SCL de la placa Arduino. En el caso de utilizar la comunicación serie, además del cable de alimentación (5V) y tierra, se ha de usar un cable conectando el pin “DO” del módulo y el pin RX de la placa Arduino, y otro conectando el pin “DI” del módulo y el pin TX de la placa Arduino. Para conectar los altavoces (se pueden acoplar hasta dos a la vez), este módulo ofrece dos parejas de terminales + y – que aportan una potencia de hasta 3 W por altavoz.

Esta placa reconoce diferentes comandos, tales como reproducir la siguiente o anterior canción, pausar o continuar la reproducción o cambiar el volumen. Dependiendo del tipo de conexión (serie o I^2C), estos comandos son diferentes y su envío se debe escribir en nuestro sketch de diferente manera. Recomiendo en este sentido consultar su documentación online para conocer los detalles. Lo que sí que hay que tener presente es que esta placa solo funcionará si existe una carpeta llamada “sound” en la raíz de la tarjeta SD, y si dentro de ella los ficheros existentes tienen las extensiones "wma", "wav", "mid" o "mp3".

Otro módulo con zócalo microSD (el cual solo admite tarjetas formateadas en FAT16) y conectores para altavoces es el “SOMO-14D” de 4DSystems. Esta placa tiene la particularidad de que solo es capaz de reproducir ficheros en formato ADPCM (.ad4) a 32 KHz y 4 bits (los cuales, además, han de estar obligatoriamente grabados directamente en la raíz de la tarjeta). Para convertir nuestros ficheros WAV o MP3 en este formato, desde la página web del producto podemos descargarnos gratuitamente un programa conversor (para Windows tan solo, no obstante).

Lo interesante de este módulo es que puede funcionar en dos modos. En el "serial mode", operaciones como "reproducir", "pausar", "parar" o "cambiar volumen" pueden ser ordenadas por la placa Arduino a través de comandos hexadecimales de 16 bits enviados por el canal serie. En el "key mode", la placa puede funcionar de forma autónoma sin necesidad de ninguna placa Arduino que la controle: tan solo se requiere un circuito formado por tres pulsadores, una batería de 3 V y un altavoz convenientemente conectados (el circuito concreto se muestra en el datasheet del producto). Si usamos el "serial mode", las conexiones son:

Placa breakout	Exterior
Nº 1	-
Nº 2	-
Nº 3 (CLK)	Una salida digital de Arduino
Nº 4 (DATA)	Una salida digital de Arduino
Nº 5	*
Nº 6	-
Nº 7	-
Nº 8	Pin 3V3 de Arduino
Nº 9	Pin GND de Arduino
Nº 10 (RESET)	Una salida digital de Arduino
Nº 11	Terminal de altavoz (8 Ω,1 W)
Nº 12	Terminal de altavoz (8 Ω,1 W)
Nº 13	-
Nº 14	-

Todos los pines del módulo trabajan a 3 V por lo que para no dañarlo necesitamos colocar resistencias de 470 Ω en serie en los conectores nº 3, 4, 5 y 10 de manera que el nivel de 5 V de las salidas de la placa Arduino se adapte convenientemente.

El pin nº 5 de la placa breakout se puede conectar opcionalmente a un pin digital de la placa Arduino configurado como entrada, y a la vez, al ánodo de un LED (cuyo cátodo se conectará a tierra a través de un divisor de tensión). Este pin emitirá una señal HIGH mientras se esté reproduciendo algún fichero de audio, por lo que si eso ocurre, el LED se encenderá y la placa Arduino detectará ese valor para poderlo tener en cuenta.

Para controlar este módulo no disponemos de ninguna librería específica, pero podemos utilizar como referencia los códigos de ejemplo disponibles en http://bit.ly/bricotuto-somo14d, los cuales aportan diferentes funciones Arduino

para las acciones más habituales, como reproducir una canción concreta (*playSong(nº cancion);*), pausarla (*pausePlay();*), reproducir la siguiente canción (*nextPlay();*), incrementar o disminuir el volumen (*incVol();* y *decVol();*), etc. En cualquier caso (también en el “key mode”), se recomienda la ayuda de un amplificador externo.

Existe un módulo muy parecido al “SOMO-14D” comercializado por Sparkfun con el código de producto nº 11125. Como novedad incluye un conector JST para enchufar una batería LiPo, pero por lo demás es funcionalmente idéntico.

Otro módulo más es el llamado “SmartWAV” de Vizictechnologies. Este es el módulo que ofrece más calidad de sonido de los hasta ahora nombrados. Concretamente, es capaz de emitir en estéreo, a 16 bits de resolución y a 48 KHz de frecuencia de muestreo (es decir, calidad CD). Además, incorpora un potenciómetro digital (el chip AD5206) que permite el control del volumen en 255 pasos y la reproducción a distintas velocidades. Por otro lado, es capaz de reconocer tarjetas tanto microSD como microSDHC, formateadas tanto en FAT16 como en FAT32, por lo que los nombres de los ficheros pueden ser más largos de ocho caracteres; y también tiene capacidad para gestionar varios niveles de carpetas. Respecto a las conexiones de audio, dispone de un conector jack de 3,5 mm para enchufar unos auriculares o un altavoz o un amplificador externo.

Igual que el “SOMO-14D”, el “SmartWAV” tiene dos modos de funcionamiento: “modo serie” y “modo autónomo”. En el modo serie operaciones como “reproducir”, “pausar”, “parar” o “cambiar volumen” pueden ser ejecutadas por la placa Arduino mediante funciones muy sencillas (del tipo *objetoSwav.playTrack()* u *objetoSwav.stopTrack()*) pertenecientes a una librería propia llamada “SmartWAV”. En realidad, esta librería (descargable de la web del producto) lo que hace es ocultar los detalles internos de la comunicación entre placa Arduino y módulo, la cual se establece a través del canal serie.

En el modo autónomo, el módulo puede funcionar sin necesidad de ninguna placa Arduino que la controle: tan solo se requiere un circuito formado por cinco pulsadores (para las señales de Reproducir/Pausa, Siguiente, Rebobinado, Volumen+ y Volumen-) y una batería de 3 V convenientemente conectados (el circuito concreto se muestra en la web del producto). Para que la placa funcione en modo autónomo, es necesario que su pin “MODE” esté conectado a tierra: si no lo está estaremos usando el “modo serie”, en cuyo caso las conexiones necesarias son las siguientes:

Placa breakout	Exterior
GND (superior izquierda)	Pin GND de Arduino
3V3 (superior izquierda)	Pin 3V3 de Arduino
TX	Pin RX de Arduino
RX	Pin TX de Arduino
RST	Pin RESET de Arduino
A	*

El pin A del módulo tiene idéntica función y conexiones que el pin nº 5 del módulo “SOMO-14D”, explicado en párrafos anteriores. Los pines GND y 3V3 a la derecha de la placa están reservados para la comunicación FTDI.

Otro módulo digno de mención es el “Embedded MP3 Module” de OpenElectronics, que puede ser controlado mediante el envío de determinados comandos propios a través del canal serie (aunque también puede funcionar en modo autónomo sin necesidad de microcontrolador si diseñamos el circuito adecuado). Los esquemas de conexiones y la lista concreta de comandos se pueden encontrar en http://www.open-electronics.org/embedded-mp3-module.

Finalmente, otro módulo a destacar es el “MP3 Trigger”, distribuido por Sparkfun con el código de producto 11029. Esta plaquita almacena en una tarjeta SD (formateada en FAT16 o FAT32) una serie de ficheros MP3 (a 192 Kbps) que se han de llamar obligatoriamente de la forma TRACK001.mp3, TRACK002.mp3 hasta TRACK256.mp3 como máximo. Dispone de una salida de audio en forma de jack de 3,5 mm para poder conectar altavoces o auriculares o un amplificador externo. Para controlarla desde Arduino, su pin “USBVCC” se ha de conectar al pin de 5 V de Arduino (para recibir la alimentación), su pin “GND” se ha de conectar al pin GND de Arduino (para conectar a tierra), su pin “RX” se ha de conectar al pin TX de Arduino y su pin “TX” al pin RX de Arduino. Puede ser controlado mediante el envío de determinados comandos propios a través del canal serie consultables en el datasheet del producto, pero afortunadamente existe una librería (descargable en https://github.com/sansumbrella/MP3Trigger-for-Arduino）, que nos facilita mucho las cosas, ya que se encarga de gestionar tanto la comunicación serie entre plaquita y Arduino como de las funciones de reproducción de los ficheros MP3 (las cuales son muy sencillas, de tipo *objetoTrigger.setVolume()* o *objetoTrigger.stop()* por ejemplo).

Lo interesante del “MP3 Trigger” es que puede ser utilizado sin intermediación de ninguna placa Arduino gracias a que dispone de hasta 18 parejas de conectores (uno para la señal de entrada y otro para tierra) que permiten poderle acoplar cualquier tipo de sensor o pulsador, de tal forma que al recibir una

determinada señal de ellos se reproduzca automáticamente un determinado fichero MP3. Para aprender a utilizar este modo de funcionamiento, remito a la documentación disponible en la página web del producto en Sparkfun.

Reproductores de voz

Sparkfun distribuye como producto nº 10661 el llamado “VoiceBox Shield”, el cual incorpora el chip SpeakJet de Magnevation. Este chip es un sintetizador de sonidos y de voz, y el “VoiceBox Shield” lo utiliza para enviarle a través del canal serie una serie de comandos propios que acaban transformándose en una nítida voz robótica mediante la conexión de un altavoz a alguno de sus pines de salida. El vocabulario es infinito porque funciona a partir de fonemas y sonidos sintetizados, y la combinación concreta de comandos hace que se varíe el tono, la velocidad, el volumen del audio generado, etc. Además del chip SpeakJet, este shield también incorpora un chip amplificador de audio interno junto con un potenciómetro (manipulable mediante un pequeño destornillador) para regular el volumen, un conector jack de audio estándar de 3,5 mm para conectar un altavoz y también un par de conectores etiquetados como SPK+ y SPK- para enchufar asimismo un altavoz o un amplificador externo.

El envío por parte de la placa Arduino de los sonidos que ha de sintetizar el shield se realiza por el canal serie, pero debido al diseño del shield, este envío no se puede realizar por el pin TX hardware de la placa Arduino, sino que ha de transmitirse por su pin digital nº 2. Por tanto, se debe utilizar la librería SoftwareSerial para comunicarse con el shield. Una ventaja de esto es que los pines serie por hardware (nº 0 y nº 1) quedan libres para otro uso.

Ejemplo 6.39: Probemos su funcionamiento con un ejemplo simple, donde se reproduce la palabra “hello”:

```
#include <SoftwareSerial.h>
/* El pin 2 envía los datos al módulo. El pin 3
no se usa (no se reciben datos) pero hay que ponerlo. */
SoftwareSerial miserie(2,3);
void setup(){
      miserie.begin(9600);
/*El array "frase" contiene un conjunto de códigos numéricos
correspondientes a fonemas (ingleses) que tiene pregrabados el chip.
Colocándolos unos tras otros podemos formar las palabras y frases que
deseemos. Concretamente, el valor del array en este ejemplo hace que
el shield reproduzca (a través de un altavoz) la palabra "hello".
```

```
También se han introducido códigos para definir la velocidad o el
tono. Para conocer los distintos fonemas asociados a los códigos
numéricos, se recomienda consultar el datasheet. */
       char frase[] = {20,96,21,114,22,88,23,5,183,7,159,146,164,0};
/*El valor final del array siempre ha de ser 0 para indicar que se ha
llegado al final de la palabra o frase a pronunciar. */
       miserie.print(frase);
}
void loop(){}
```

Concretamente, para decir la palabra “hello” aproximadamente, los fonemas necesarios son el código 183 (sonido “h”), el 159 (sonido “eh”), el 146 (sonido “lu”) y el 164 (sonido “oh”). Otros códigos son el 20,96 (para establecer el volumen hasta nueva orden; el volumen mínimo es 20,0 y el máximo es 20,127); el 21,114 (para establecer la velocidad); el 22,88 (para establecer el tono de voz a 88Hz); el 23,5 (para establecer el timbre de la voz; para un sonido profundo sería 23,0 y para un sonido agudo y metálico sería 23,15); el 7 (para reproducir el siguiente fonema el doble de rápido que usualmente), el 4 (para añadir una breve pausa entre fonemas), etc. Conjugando correctamente los valores de tono y velocidad, incluso podríamos hacer que el chip cantara.

A partir de aquí, no debería ser difícil escribir un sketch Arduino que, dependiendo de la activación de pulsadores u otro tipo de entradas provenientes de diferentes sensores, emitiera una frase u otra. Existe un software (solo para sistemas Windows) desarrollado por la misma empresa fabricante del chip Speakjet llamado Phrase-A-Lator que contiene un extenso diccionario de transcripciones fonéticas de palabras del idioma inglés, listas para ser usadas en nuestro código Arduino. Pero lo más interesante es que permite obtener los códigos equivalentes a las palabras deseadas de una forma muy intuitiva.

Otra alternativa por si no queremos aprender los códigos numéricos del chip Speakjet es utilizar en conjunción con este el chip TTS256. Este chip actúa como un intermediario entre el usuario y el chip Speakjet, ya que permite recibir a través del canal serie una cadena de texto literal (en inglés, eso sí) para convertirla en los códigos adecuados y entonces pasárselos al chip Speakjet. De hecho, el “SpeakJet Shield TTS” de http://www.droidbuilder.com es un shield bastante similar al VoiceBox de Sparkfun pero con la ventaja de que incorpora ambos chips, por lo que su programación es realmente sencilla ya que tan solo es necesario el uso de un objeto SoftwareSerial y su función *print()*. Desgraciadamente, solo se distribuye en forma de kit, por lo que es necesario soldar siguiendo las instrucciones indicadas en su página.

Una competencia directa al "VoiceBox Shield" de Sparkfun es el "GinSing Shield" de GinSingSound. Este shield (que se puede adquirir completamente ensamblado o en forma de kit) viene con el chip sintetizador Babblebot, el chip amplificador NJM386 y (al igual que el producto de Sparkfun) dispone de un conector de salida de audio de tipo jack de 3,5 mm para poder enchufar unos auriculares.

Ambos shields son bastante parecidos, aunque el chip SpeakJet está más enfocado a facilitar la síntesis del habla y el chip BabbleBot es más bien un generador de sonidos complejos, que permite la creación de efectos, la reproducción de hasta seis canales de audio a la vez, la síntesis de música y voz, etc. De hecho, el chip BabbleBot que viene de fábrica en el shield GinSing está preconfigurado para trabajar en cada momento en un modo de funcionamiento de entre cuatro modos posibles: activación de efectos sonoros predefinidos o creados por uno mismo ("preset mode"), creación de un instrumento musical polifónico ("poly mode"), generación del habla ("voice mode") y ejecución de síntesis de una onda de audio ("synth mode").

Sea cual sea el modo de funcionamiento que queramos utilizar (los cuales podemos ir cambiando a lo largo de nuestro sketch sin problemas), primero es necesario descargar e instalar una librería propia (la "GinSing Lib") para poder interactuar convenientemente con el shield GinSing. Está disponible en el apartado de descargas de la página http://www.ginsingsound.com .

Centrándonos en el uso de la generación del habla ("voice mode"), podemos consultar la lista de fonemas admitidos dentro de la sección titulada "GSAllophone" del fichero GinSingDefs.h, incluido dentro de la librería anterior. A partir de aquí, podemos estudiar los diferentes códigos de ejemplo incluidos también dentro de la librería anterior, que sirven para mostrar las funcionalidades de cada uno de los modos de funcionamiento del chip. Concretamente, el completo (y largo) código de ejemplo llamado "4_voicemode.ino" nos permite reproducir diferentes frases, incluso cantadas, y aprender el uso del "voice mode" del shield.

Otro shield "parlante" diferente es el llamado "Voice shield" de Spikenzielabs, el cual se basa en el chip ISD4003 de Winbond. En este caso, el sonido no se sintetiza sino que se ha de grabar previamente. Lo interesante de este shield es que permite grabar diferentes unidades de audio identificándolas con un número entero, de forma que podamos combinarlas de la manera que queramos para formar cadenas de unidades más complejas. Es decir, podemos grabar por ejemplo de forma separada la palabra "Yo" (con identificador nº 1), la palabra "Hablo" (con identificador nº 2) y la palabra "Miro" (con identificador nº 3), y entonces hacer que el shield reproduzca la frase "Yo hablo" indicando las unidades 1 y 2 o la frase "Yo miro" indicando las

unidades 1 y 3. El sonido emitido, no obstante, tan solo tiene una frecuencia de muestreo de 8 KHz y el total de unidades no puede exceder de 4 minutos (240 segundos). Para reproducir el sonido guardado, se puede enchufar un altavoz o amplificador externo a un conector jack de 3,5 mm etiquetado como “Audio OUT”, pero también se puede utilizar un amplificador interno ya integrado en el shield junto con un altavoz de 15 mm a soldar en un espacio reservado.

La manera más sencilla de introducir las unidades de audio en este shield es la siguiente: primero debemos tener guardados en una determinada carpeta de nuestro computador un conjunto de ficheros (en formato WAV o MP3) que representan cada una de las unidades que queremos, y además tener un fichero de texto donde cada una de sus líneas ha de seguir el siguiente formato: *identificador de unidad|tabulador|nombre con extensión del fichero de sonido* (¡este nombre no ha de tener espacios!). A partir de aquí, deberemos descargarnos e instalar una librería propia llamada “VS Arduino Library”, disponible en la página web del producto. Seguidamente, con el shield acoplado a nuestra placa Arduino, deberemos abrir y grabar en ella un sketch (descargado junto con la librería) llamado “VSLoader.ino”. Esto hará que nuestra placa Arduino sea capaz de recibir correctamente la lista de unidades de audio para grabarlas en el chip ISD4003 del shield.

Para realizar esa grabación, nuestra placa Arduino ha de mantenerse conectada vía USB a nuestro computador para controlar el proceso pero además, la entrada de audio del shield (concretamente, un conector jack de 3,5 mm etiquetado como “Audio IN”) ha de conectarse mediante el cable adecuado a una salida de audio de nuestro computador para recibir los ficheros de sonido propiamente dichos. Una vez realizadas estas conexiones, debemos cerrar el entorno de desarrollo Arduino y ejecutar un programa multiplataforma (descargable de la página web del producto) llamado “VSProgrammer”. En él deberemos indicar la carpeta donde se ubican los ficheros de audio que representan las unidades, y el fichero de texto con la lista de todos ellos. Clicando en el botón “Program” se realizará la grabación. Una vez ya grabados, para manejar estos sonidos en nuestros sketches deberemos usar las funciones que proporciona para ello la librería “VS Arduino Library”.

7 SENSORES

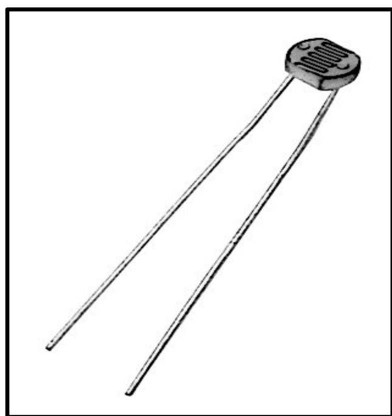
Más divertido que hacer parpadear LEDs es utilizar sensores de todo tipo para detectar qué está pasando "ahí fuera" y reaccionar en consecuencia. Desgraciadamente, cada sensor tiene sus propios métodos de conexión: algunos necesitan resistencias "pull-up" y otros no, algunos necesitan fuentes de alimentación propias y otros no, algunos trabajan a mucha tensión y otros no, etc. En este capítulo se presentarán los sensores más comunes, con ejemplos de circuitos donde se usan y código Arduino que los hacen funcionar.

También se indicará para cada tipo de sensor específico qué productos concretos podemos encontrar en diferentes distribuidores. De todas formas, si se desea, se puede adquirir cómodamente de una sola vez un conjunto de diversos sensores gracias al "Sensor pack 900" de Adafruit (código de producto nº 176) o el "Sensor Kit" de Sparkfun (código de producto 11016). El primero incluye un LED infrarrojo y un sensor infrarrojo específicos para control remoto, un sensor de luz, un sensor de temperatura, un sensor de inclinación, un sensor de golpes (usable como zumbador), un sensor de campo magnético (junto con un imán), un sensor de fuerza y un acelerómetro. El segundo incluye un sensor infrarrojo específico para control remoto, un sensor de luz, un sensor de flexión, un sensor de golpes y vibraciones, un sensor de campo magnético (junto con un interruptor sensible a él –lo que se llama un "reed switch"–), un sensor de fuerza, un sensor de humedad, un sensor de distancia, un sensor de movimiento, un acelerómetro, un giroscopio, una brújula (un magnetómetro) y un sensor de presión atmosférica (un barómetro). Además, incluye un potenciómetro de membrana fina con recorrido lineal (producto nº 8680).

Otro kit de sensores interesante es el ofrecido por Cutedigi con código de producto H21, el cual contiene un sensor de temperatura, de humedad, de sonido, de efecto Hall, de inclinación, de obstáculos, de fuego, de metal, un acelerómetro, una brújula, un LDR, un "reed switch"... además de un emisor y receptor de infrarrojos, un pulsador, un zumbador, un LED RGB, un optointerruptor, y más.

SENSORES DE LUZ VISIBLE

Fotorresistores



Los sensores de luz, tal como su nombre indica, son sensores que permiten detectar la presencia de luz en el entorno. A veces se les llama "celdas CdS" (por el material con el que suelen estar fabricados, sulfuro de cadmio) o también "fotorresistores" y LDRs (del inglés "Light Dependent Resistor"), ya que básicamente se componen de una resistencia que cambia su valor dependiendo de la cantidad de luz que esté incidiendo sobre su superficie. Concretamente, reducen su resistencia a medida que reciben más intensidad de luz.

Suelen ser pequeños, baratos y fáciles de usar; por esto aparecen mucho en juguetes y dispositivos domésticos en general. Pero son imprecisos: cada fotorresistor reacciona de forma diferente a otro, aunque hayan sido fabricados en la misma tongada. Es por eso que no deberían ser usados para determinar niveles exactos de intensidad de luz, sino más bien para determinar variaciones en ella, las cuales pueden provenir de la propia luz ambiente ("amanece o anochece") o bien de la presencia de algún obstáculo que bloquee la recepción de alguna luz incidente. También se podría tener un sistema de fotorresistores y comparar así cuál de ellos recibe más luz en un determinado momento (para construir por ejemplo un robot seguidor de caminos pintados de blanco en el suelo o de focos de luz, entre otras muchas aplicaciones).

Otro dato que hay que saber es que su tiempo de respuesta típico está en el orden de una décima de segundo. Esto quiere decir que la variación de su valor resistivo tiene ese retardo respecto los cambios de luz. Por tanto, en circunstancias donde la señal luminosa varía con rapidez su uso no es muy indicado (aunque

también es verdad que esta lentitud en algunos casos es una ventaja, porque así se filtran variaciones rápidas de iluminación).

A la hora de adquirir un fotorresistor hay que tener en cuenta además otra serie de factores: aparte del tamaño y precio, sobre todo hay que mirar también la resistencia máxima y mínima que pueden llegar a ofrecer. Estos datos lo podremos obtener del datasheet que ofrece el fabricante. De hecho, en el datasheet no solo podemos consultar estos dos datos extremos sino también todos los valores intermedios de resistencia, gracias a un conjunto de gráficas que nos indican cómo varía de forma continua (y generalmente logarítmica) el valor resistivo del fotorresistor en función de la cantidad de luz recibida, medida en unidades lux. Con esta información podemos conocer, sabiendo la luz que incide sobre el LDR, qué valor de resistencia ofrece este (y viceversa: sabiendo la resistencia que ofrece, podemos deducir la cantidad de luz que recibe el sensor).

En el párrafo anterior se menciona un “conjunto de gráficas” y no una sola porque el comportamiento del fotorresistor depende de la temperatura ambiente: según sea ésta, la variación de la resistencia respecto la luz incidente será una u otra. También se menciona que esta variación es “generalmente logarítmica”; es posible que haya casos donde no lo sea, pero lo que es seguro es que, por la propia naturaleza intrínseca de los fotorresistores, la relación entre cantidad de luz recibida y resistencia resultante nunca será lineal. Esto significa, por ejemplo, que si la intensidad lumínica es de 10 lux y se mide una resistencia de 100 Ω, cuando esta sea 20 lux la resistencia no tendrá por qué ser de 50 Ω.

Otro dato a consultar en el datasheet para tenerlo en cuenta es la sensibilidad. Los fotorresistores no detectan del mismo modo los diferentes tipos de luz; concretamente, suelen ser más sensibles a cambios en luces de color verde que en los de otros colores. Además, existen una longitudes de onda mínimas (400 nm, normalmente) y máximas (600 nm, normalmente) más allá de las cuales no detectan nada. Esta información la podemos encontrar en forma de gráfica que muestra la respuesta del fotorresistor en función de la longitud de onda recibida.

Breve nota sobre el espectro electromagnético:

Llamamos “luz” a las ondas que son de tipo electromagnético. Por tanto, como ondas que son, una de sus características que podemos estudiar es su frecuencia –medida en Hz– o, alternativamente, su longitud de onda –medida en nanómetros (10^{-9} metros)–. Ambas magnitudes están relacionadas por la expresión $\lambda = c/\nu$

(donde λ representa la longitud de onda en el vacío, ν la frecuencia y c la velocidad de la luz en el vacío, que es una constante igual a 299.792.458 m/s).

Podemos clasificar diferentes tipos de luz según su longitud de onda: así tenemos los rayos gamma y los rayos X (con menor longitud de onda), pasando de forma continua (¡la luz es una magnitud analógica!) por la luz ultravioleta, la luz visible y los rayos infrarrojos, hasta llegar a las ondas electromagnéticas de mayor longitud de onda como son las ondas de radio. El conjunto de todas estas ondas es lo que se llama el espectro electromagnético

El ojo humano no es capaz de percibir todo el espectro electromagnético: solo una pequeña parte identificada como "luz visible". Esto significa que en nuestro día a día, realmente estamos rodeados de radiación electromagnética que no vemos. Aunque no hay límites exactos para la zona visible del espectro (depende de cada persona), se suelen tomar como valores aceptados las ondas que tengan una longitud de onda entre los 400 nm y 700 nm.

Dentro del espectro visible, dependiendo de la longitud de onda concreta que tenga una determinada onda, esta se verá de un color u otro. Así, podemos decir aproximadamente que la luz visible de longitud de onda entre 400 y 450 nanómetros es de color violeta, entre 450 y 495 de color azul, entre 495 y 570 de color verde, entre 570 y 590 de color amarillo, entre 590 y 620 de color anaranjado y entre 620 y 700 de color rojo.

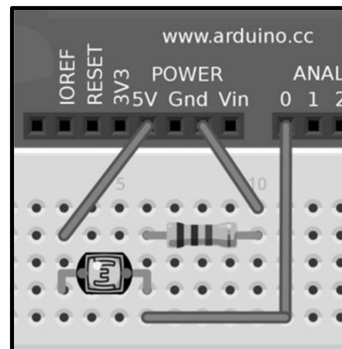
Por otro lado, la tensión aceptada por estos dispositivos puede ser prácticamente cualquiera (hasta los 100 V). Debido a que los fotorresistores no son más que resistencias, no están polarizados, así que sus terminales se pueden conectar en nuestros circuitos en ambos sentidos.

La manera más fácil de comprobar que un sensor de luz funcione es conectar sus terminales a un multímetro en modo medida de resistencia y hacerle incidir más o menos luz. Si vemos que responde (hay que vigilar con el cambio de escala), ya podremos empezar a diseñar nuestros proyectos con él.

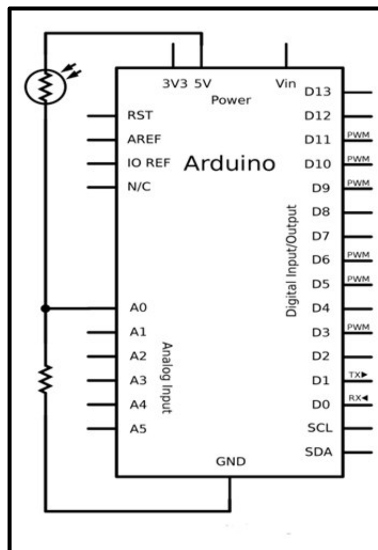
Lo primero que hemos de saber es cómo se conecta un sensor de luz a nuestro circuito. Lo que haremos será conectar uno de los terminales del sensor a la alimentación y el otro, a través de una resistencia "pull-down" (sobre cuyo valor resistivo adecuado discutiremos unos párrafos más abajo), a tierra. Además, desde un punto entre el fotorresistor y la resistencia "pull-down" conectaremos un "tercer cable" hacia una entrada analógica de nuestra placa Arduino para que esta pueda leer el voltaje analógico a medir. Este voltaje recibido podrá oscilar entre 0 V y el

voltaje que alimente al fotorresistor: en las figuras siguientes mostraremos el fotorresistor alimentado con los 5 V del pin "5V" de la placa Arduino, pero también podría alimentarse perfectamente con los 3,3 V del pin "3V3", por ejemplo.

Si usamos una resistencia "pull-down", cuanto mayor voltaje recibamos por la entrada analógica de la placa Arduino significará que más luz incide en el sensor. Si hubiéramos utilizado una resistencia "pull-up", sería al revés: a mayor voltaje recibido significaría que hay más oscuridad. Nosotros, tal como hemos comentado, utilizaremos una resistencia "pull-down", por lo que, en definitiva, el montaje sería similar a este:



Y el esquema eléctrico a este (aquí se puede apreciar un símbolo para identificar el fotorresistor, no visto hasta ahora):



El truco para entender en profundidad este montaje es ver que a medida que la resistencia del fotorresistor va decreciendo (porque le incide más luz), la resistencia total del conjunto de resistencias en serie fotorresistor + pull-down también decrece. Por la Ley de Ohm, esto hará que (al mantenerse un voltaje de alimentación fijo en todo el circuito –de 5 V–) la intensidad de corriente aumente a través de todo ese circuito. Pero como la resistencia "pull-down" es fija, por la misma Ley de Ohm, si la intensidad que la atraviesa ha aumentado, también lo habrá hecho el voltaje entre sus terminales. Que es de hecho lo que medimos con el "tercer cable": el voltaje existente entre los terminales de la resistencia "pull-down". Si la resistencia del fotorresistor fuera despreciable –al haber mucha luz–, ese voltaje medido sería 5 V; si la resistencia del fotorresistor fuera tan grande que abriera el circuito interrumpiendo el paso de electrones –al haber mucha oscuridad–, el voltaje medido sería 0 V. Entre ambos casos extremos tendremos medidas intermedias.

Lo explicado en el párrafo anterior se puede resumir en la siguiente fórmula, obtenida a partir de la Ley de Ohm y del hecho que la intensidad que atraviesa ambas resistencias es la misma: $\mathbf{V_{med} = (R_{pull} / (R_{pull} + R_{foto}))\cdot V_{fuente}}$, donde V_{fuente} es el voltaje aportado por la fuente de alimentación, V_{med} es el voltaje recibido por el pin de entrada analógico (es decir, el existente entre los terminales de la resistencia "pull-down", el cual puede valer entre 0 V y V_{fuente}), R_{pull} es el valor de la resistencia "pull-down" (fijo) y R_{foto} es el valor de la resistencia del fotorresistor. De aquí se puede comprobar lo que ya hemos dicho: que cuando aumenta la intensidad de luz (como la resistencia del fotorresistor disminuye), también aumenta el voltaje medido, y viceversa.

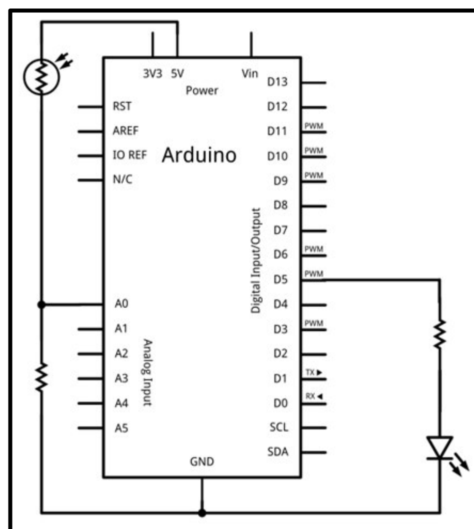
No obstante, en realidad V_{med} no es el valor con el que trabajamos en nuestra placa Arduino, porque esta utiliza siempre un conversor analógico-digital para realizar un mapeo de todos valores analógicos recibidos (los cuales pueden oscilar entre 0 V y 5 V –suponiendo que el voltaje proporcionado por la fuente son 5 V–) a valores digitales (que van entre 0 y 1023). Estos valores digitales son los que la placa Arduino entiende en realidad y con los que trabajaremos en nuestros sketches. La buena noticia es que la conversión de valores analógicos a digitales se puede expresar por una simple regla de proporcionalidad así: $\mathbf{V_{convertido} = V_{med} \cdot 1023/5}$. A partir de aquí, si sustituimos esta expresión en la fórmula del párrafo anterior, y despejamos de allí R_{foto}, llegamos a la expresión siguiente: $R_{foto}=(R_{pull}\cdot 1023/V_{convertido}) - R_{pull}$, la cual nos permite conocer por fin cuál es el valor actual de la resistencia del fotorresistor a partir del voltaje digitalizado obtenido por la placa Arduino.

El paso natural siguiente, una vez conocido el valor de la resistencia del fotorresistor, sería averiguar a qué cantidad de iluminación se corresponde. Esto se

puede consultar en las gráficas del datasheet, tal como se ha comentado previamente. No obstante, este paso no suele hacerse ya que normalmente los fotorresistores se utilizan para comparar iluminaciones (entre diferentes sitios o entre distintos intervalos de tiempo) más que para obtener valores concretos de iluminación.

Para saber el valor adecuado de la resistencia "pull-down" que tenemos que colocar en nuestro circuito, tendríamos que conocer (a partir del datasheet) los distintos valores numéricos concretos que puede adquirir nuestro fotorresistor a lo largo de distintas iluminaciones (R_{foto}) y utilizarlos, junto con el valor del R_{pull} obtenido, en la fórmula ya vista en la página anterior $V_{med} = (R_{pull} / (R_{pull} + R_{foto}))\cdot V_{fuente}$ para observar qué V_{med} hipotético obtendríamos. Haciendo esto, veremos que en la gran mayoría de los casos, el comportamiento de R_{foto} respecto a la iluminación recibida hace que valores elevados de R_{pull} (por ejemplo, 10 KΩ) saturen rápidamente las lecturas en entornos brillantes. Es decir, hace que el sensor llegue a medir el tope de los 5 V con una iluminación relativamente baja y no sea por tanto capaz de distinguir entre un ambiente bien iluminado de otro muy bien iluminado. En cambio, valores menores de R_{pull} (como por ejemplo, 1 KΩ), sí permitirán detectar cambios en la luz más brillante pero harán que no sea posible distinguir diferencias en niveles oscuros. Por lo tanto, dependiendo del entorno donde situemos nuestro proyecto, deberemos elegir una R_{pull} de 10 KΩ (para entornos oscuros) o de 1 KΩ (para entornos iluminados), o bien utilizar algún tipo de potenciómetro ajustable.

Ejemplo 7.1: Comprobemos ya cómo se comporta un fotorresistor cuando lo conectamos a una placa Arduino. Para ello podemos montar el siguiente circuito; fijarse que el LED se ha de conectar a un pin PWM.



Lo que queremos es utilizar el valor del voltaje leído en el pin de entrada analógico (el nº 0 en este caso) para iluminar consecuentemente el LED: cuanto menos luz detecte el fotorresistor, más brillante iluminará el LED (por eso es necesario que el LED reciba una señal analógica también, a través de un pin PWM, el nº 5 en este caso). Además, se muestra por el canal serie los valores obtenidos por el fotorresistor.

```
int valorcds;   //Valor obtenido
int brilloLED;  //Valor enviado al LED
void setup(void) {
       Serial.begin(9600);
}
void loop(void) {
       valorcds = analogRead(0);
/*Además de imprimir "valorcds" tal cual, también se podría haber
comprobado si este es menor o mayor que una cantidad dada, y haber
imprimido un mensaje tal como "Oscuro", "Normal", "Brillante", etc*/
       Serial.println(valorcds);
/*El valor obtenido "valorcds" será mayor cuanto más brillante sea el
entorno. En cambio, el LED ha de iluminar más cuanto más oscuro sea
el entorno. Es decir, "brilloLED" ha de ser mayor cuanto menor sea el
valor de "valorcds". Por eso, hemos de invertir "valorcds" para que
su valor pase de una escala de 0 a 1023 a otra de 1023 a 0.*/
       valorcds=1023-valorcds;
/*Y ahora, tal como ya hemos visto en ejemplos anteriores, hemos de
mapear "valorcds" para que caiga dentro del rango admitido para la
salida PWM. Es decir, pasar un valor que está entre 0 y 1023 ha otro
que está entre 0 y 255.*/
       brilloLED = map(valorcds, 0, 1023, 0, 255);
       analogWrite(5, brilloLED);
       delay(100); //Para que se pueda ver el nuevo brillo
}
```

Con lo ya sabido, podríamos realizar un simple detector de presencia. Si mantuviéramos iluminado el fotorresistor de forma constante, al interponerse algún obstáculo entre la fuente de luz y el fotorresistor, este detectaría una bajada brusca de intensidad lumínica. Usando el mismo circuito del ejemplo anterior, podríamos modificar levemente el sketch para enviar una señal digital de salida al LED, de forma que esta valiera HIGH (encendiendo el LED) si el fotorresistor detectara un valor por debajo de un cierto valor umbral elegido por nosotros (y por tanto, detectara que alguien se interpone entre la luz y el sensor) o que valiera LOW (apagando el LED) si el valor "valorcds" fuera mayor que dicho umbral (y por tanto, se detectara una incidencia "normal" de la luz en el sensor). Se deja como ejercicio.

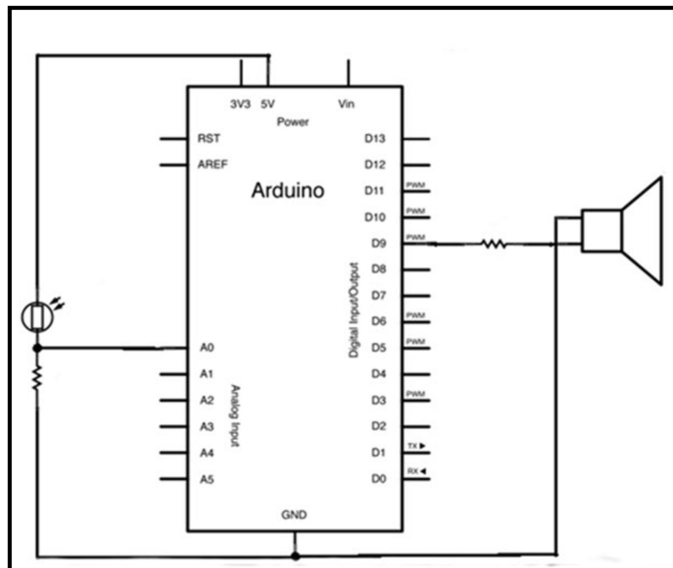
Por otro lado, ya hemos comentado antes que las lecturas obtenidas por un fotorresistor como los que usamos en nuestros proyectos (por ejemplo, el producto nº 9088 de Sparkfun o el nº 161 de Adafruit) no suelen ser muy precisas. Es muy recomendable, por tanto, calibrar estos componentes antes de empezar a trabajar con ellos (por ejemplo, dentro de la función "setup()". Calibrar significa fijar el valor mínimo y máximo del rango de posibles valores a leer si ya sabemos que estos no llegan nunca a 0 o a 1023, respectivamente. De esta manera, se pueden interpretar mejor las lecturas realizadas porque los valores intermedios son más coherentes.

Ejemplo 7.2: El siguiente sketch pretende calibrar un fotorresistor, aunque el procedimiento es generalizable sin apenas cambios a cualquier otro sensor analógico. El circuito necesario es el mismo que el de los ejemplos anteriores: un fotorresistor conectado a un pin de entrada analógico (supondremos el 0) acompañado de una resistencia "pull-down" de 10 KΩ, y un LED conectado a un pin de salida PWM (supondremos el 5) acompañado de su divisor de tensión de 220 Ω correspondiente. El sketch lo que hace básicamente es leer durante sus primeros cinco segundos de ejecución una serie de valores del sensor analógico para establecer cuál será su lectura con valor mínimo y cuál será la que tenga el valor máximo. Es evidente que durante esos cinco segundos, deberemos someter el sensor a ambas circunstancias extremas para ver cómo reacciona (en caso de un fotorresistor, iluminándolo con la máxima luz prevista en el proyecto y con la mínima). Una vez realizada la calibración, el resto del código es muy parecido al visto anteriormente: la clave está en la función *map(),* primero porque realiza ella misma el mapeo en rangos invertidos (por lo ya explicado de iluminar el LED cuando se detecte poca luz), pero sobre todo porque el mapeo lo establece en el rango de valores calibrados, no los típicos 0 y 1023.

```
int valorcds = 0;
int sensorMin = 1023; //Irá disminuyendo
int sensorMax = 0;    //Irá aumentando
void setup() {
     //Calibramos durante los primeros 5 segundos de programa
     while (millis() < 5000) {
           valorcds = analogRead(0);
     /*Si se lee un valor mayor que el actual máximo,
       lo guardo como el nuevo valor máximo */
           if (valorcds > sensorMax) {
                sensorMax = valorcds;
           }
     /*Si se lee un valor menor que el actual mínimo,
       lo guardo como el nuevo valor mínimo */
           if (valorcds < sensorMin) {
```

```
                sensorMin = valorcds;
            }
        }
}
void loop() {
            valorcds = analogRead(0);
/*Aplico la calibración a la lectura recién leída a la transformación
que ha de sufrir valorcds para ser usada en analogWrite() */
        valorcds = map(valorcds, sensorMax, sensorMin, 0, 255);
/*En el caso de que la lectura recién leída caiga fuera del rango
establecido durante la calibración…*/
            valorcds = constrain(valorcds, 0, 255);
            //Ilumino el LED usando el nuevo valor calibrado
            analogWrite(5, valorcds);
}
```

Ejemplo 7.3: Si tenemos el circuito mostrado en la figura siguiente (es decir, un LDR con resistencia "pull-down" de 10 KΩ y un zumbador con divisor de tensión de unos 100 Ω, por ejemplo), podemos hacer que según sea la luz detectada por el LDR, la frecuencia del sonido emitido por el zumbador vaya cambiando.

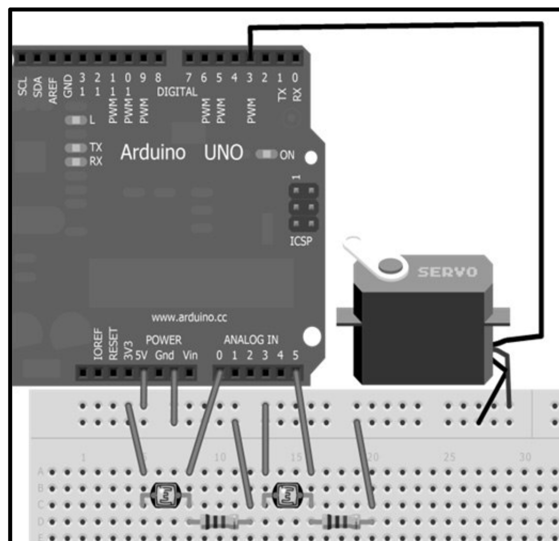


El código es este: simplemente realiza la lectura del LDR, la mapea a un rango de valores audibles y la envía al zumbador. Se debería de calibrar primero el LDR para ajustar el mapeo convenientemente, ya que los valores de *map()* del sketch son solamente orientativos.

```
void setup() {}
void loop() {
     int lectura;
     int sonido;
     lectura = analogRead(0);
/*En este caso, la calibración del LDR da valores entre 400 y 1000,
pero esto puede variar. El rango de salida son, en Hz, las
frecuencias mínimas y máximas del sonido que queremos emitir*/
     sonido = map(lectura, 400, 1000, 120, 1500);
     tone(9, sonido, 10);
     delay(1); //Por estabilidad entre lecturas
}
```

No sería demasiado difícil sustituir en el circuito anterior el LDR por un potenciómetro. De hecho, esta misma variante ya se vio en el apartado relativo al sonido del capítulo 6.

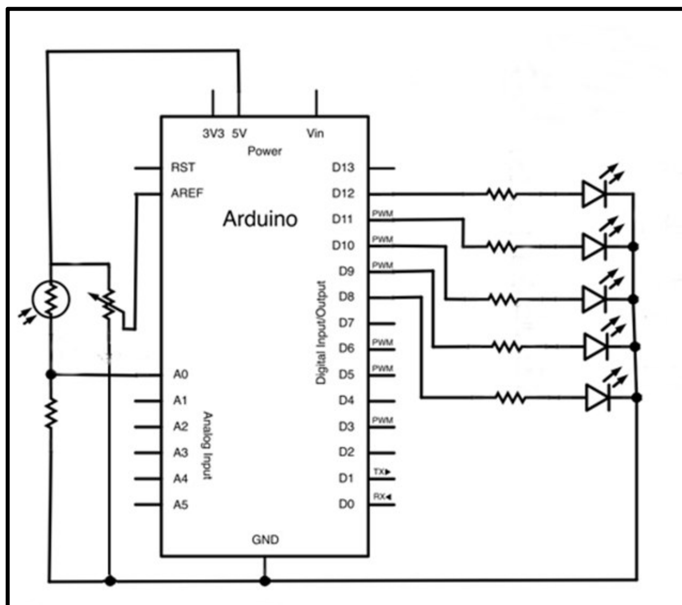
<u>Ejemplo 7.4</u>: Otro ejemplo práctico con LDRs es el diseño de un dispositivo capaz de orientarse hacia el punto más oscuro de su entorno, gracias a los valores obtenidos por dos sensores de luminosidad estratégicamente situados. La idea es comparar la lectura de ambos sensores y orientar un servomotor en un sentido u en otro según si la lectura de un sensor es mayor o menor que la del otro. El montaje sería algo parecido al siguiente:



Y el código, este:

```
#include <Servo.h>
int valorLDRderecha = 0;
int valorLDRizquierda = 0;
int angulo = 0;
Servo miservo;
void setup() {
   miservo.attach(3);
}
void loop() {
   valorLDRizquierda = analogRead(0);
   valorLDRderecha = analogRead(5);
   if (valorLDRderecha < valorLDRizquierda) {
         angulo = angulo - 10;
         if (angulo < 0) {angulo = 0;}
   } else {
         angulo = angulo + 10;
         if (angulo > 179) {angulo = 179;}
   }
   miservo.write(angulo);
}
```

Ejemplo 7.5: Como último ejemplo de demostración práctica del uso de LDRs, vamos a diseñar el siguiente circuito:



Un valor adecuado para los divisores de tensión de los LEDs puede ser 220 Ω. Un valor adecuado para la resistencia “pull-down” asociada al LDR puede ser de 1 KΩ. Un valor adecuado para la resistencia máxima del potenciómetro puede ser 10KΩ. El LDR, tal como se puede ver, está conectado al pin de entrada analógica nº 0.

La idea es iluminar nuestro entorno con LEDs a medida que este vaya oscureciendo. Pero en vez de utilizar un solo LED regulado analógicamente como vimos en uno de los ejemplos anteriores, ahora utilizaremos varios LEDs controlados por señales digitales, de manera que se iluminen más LEDs a medida que se vaya detectando más oscuridad.

La gran novedad de este circuito es el uso de un voltaje de referencia controlado por un potenciómetro. La razón de haber incluido este elemento es la siguiente: ya sabemos que la lectura obtenida del fotorresistor puede variar desde 0 (correspondiente a 0 V, cuando el ambiente está completamente a oscuras) hasta 1023 (correspondiente a 5 V, cuando el ambiente está completamente iluminado). Pero estos dos casos extremos pueden ser difíciles de conseguir, por lo que quizás solo vayamos a trabajar en un rango de 500 o 600 valores intermedios, despreciando así mucha resolución. Para solucionar esto y para permitir además que nuestro circuito se adapte a entornos con diferentes extremos de iluminación sin tener que cambiar cada vez los valores umbral escritos en nuestro código, se ha optado por utilizar un voltaje de referencia ajustable. Este voltaje marca el punto a partir del cual, si decrece la luz ambiente, empiezan a decrecer los valores obtenidos en el pin analógico de entrada. Si esta referencia es baja, los LEDs empezarán a iluminarse con poca luz ambiente pero serán más sensibles a variaciones leves de iluminación; si esta referencia es alta, los LEDs empezarán a iluminarse con mucha luz ambiente pero esta deberá variar mucho para modificar el número de LEDs iluminados. En otras palabras: el potenciómetro sirve para fijar el umbral de luz mínima, a partir del cual, comenzará a funcionar nuestro circuito de luz artificial.

Esto es así porque variando la señal de referencia, le estamos diciendo que el rango de 1024 valores estén ubicados entre 0 V y una determinada tensión máxima que será una u otra según las circunstancias. En nuestro sketch hemos dividido el rango 0-1024 en cinco secciones para que se activen progresivamente cada uno de los cinco LEDs. Si nuestro entorno ofrece una variación muy baja de iluminación con el que jugar, podemos ajustar la tensión de referencia a un valor bajo (por ejemplo, 1 V) y así nos seguirá distribuyendo proporcionalmente la activación de las salidas, con lo que conseguiremos una mayor sensibilidad. El “precio a pagar” es el no activar el circuito hasta que la entrada analógica no reciba esos 1 V, ya que mientras reciba un voltaje mayor, el conversor analógico-digital estará saturado.

Con este truco, pues, no nos será necesario realizar ningún cambio en el código si cambiamos a un entorno con cambios en la iluminación más o menos extremos: tan solo variando la posición del potenciómetro, el rango 0-1024 automáticamente se adaptará a las nuevas circunstancias lumínicas externas. El código es el siguiente:

```
int valorLDR = 0;
byte i;
void setup() {
  //Utilizaremos 5 LEDs, conectados a los pines 8,9,10,11 y 12
  for (i=8;i<=12;i++) { pinMode(i,OUTPUT); }
  analogReference(EXTERNAL);
}
void loop() {
  valorLDR = analogRead(0);
  /*Si el entorno está muy iluminado (según la referencia marcada
   por el potenciómetro), no se enciende ningún LED */
  if(valorLDR >= 1023){
         for (i=8;i<=12;i++){ digitalWrite(i,LOW); }
  //Si está algo menos, se enciende un LED
  } else if(valorLDR >= 823){
         digitalWrite(8, HIGH);
         for (i=9;i<=12;i++){ digitalWrite(i,LOW); }
  //Si está algo menos, se encienden dos LEDs
  } else if(valorLDR >= 623){
         for (i=8;i<=9;i++) { digitalWrite(i,HIGH); }
         for (i=10;i<=12;i++) { digitalWrite(i,LOW); }
  //Si está algo menos, se encienden tres LEDs
  } else if(valorLDR >= 423){
         for (i=8;i<=10;i++){ digitalWrite(i,HIGH); }
         for (i=11;i<=12;i++) { digitalWrite(i,LOW); }
  //Si está algo menos, se encienden cuatro LEDs
  } else if(valorLDR >= 223){
         for (i=8;i<=11;i++){ digitalWrite(i,HIGH); }
         digitalWrite(12,LOW);
  //Si el entorno está muy oscuro, se encienden los cinco LEDs
  } else {
         for (i=8;i<=12;i++){ digitalWrite(i,HIGH); }
  }
}
```

El sensor digital TSL2561

Además de los fotorresistores (que son sensores analógicos), también existen sensores de luz que son digitales, como por ejemplo el chip TSL2561 que Adafruit distribuye sobre una cómoda plaquita breakout. Los sensores digitales son más precisos que los fotorresistores (ya que permiten lecturas exactas, medidas en unidades lux) y su sensibilidad puede ser configurada dependiendo de la intensidad de luz con la que se trabaje en ese momento (intensidad cuyo rango admitido es además mucho más amplio que el de los fotorresistores). Además, el TSL2561 concretamente detecta, además de todo el espectro visible, también la luz infrarroja; pudiéndose configurar para medir separadamente la luz visible, la luz infrarroja o ambos.

Este chip se alimenta con un voltaje de entre 2,7 V y 3,6 V y funciona como mucho a 0,5 mA, por lo que es ideal para sistemas de bajo consumo. Su sistema de comunicación con el exterior es el protocolo I^2C, por lo que en la plaquita breakout en la que se comercializa, además de los contactos de alimentación y tierra, aparecen los contactos “SDA” (a conectar al pin analógico nº 4 de Arduino) y “SCL” (a conectar al pin analógico nº 5 de Arduino).

La exactitud de este sensor tiene un “precio”, y es la dificultad de su uso: además de la propia complejidad interna que aporta del protocolo I^2C, para deducir la cantidad de luminosidad exacta leída por el chip se han de utilizar muchos cálculos matemáticos poco intuitivos. Afortunadamente, Adafruit ofrece una librería Arduino propia que facilita mucho la obtención e interpretación de datos, descargable desde https://github.com/adafruit/TSL2561-Arduino-Library. Por falta de espacio no podemos profundizar en el uso de esta librería, pero si se quiere empezar a aprender a usarla, después de instalarla como cualquier librería Arduino, recomiendo observar el código comentado de los sketches de ejemplo que vienen junto con la librería.

El sensor analógico TEMT6000

También existen chips sensores de luz con comportamiento analógico. Un ejemplo es el TEMT6000, distribuido en forma de plaquita breakout por Sparkfun (producto nº 8688). Este sensor tiene la ventaja de ser mucho más preciso que un fotorresistor (reacciona mejor a cambios de iluminación en un rango mayor) sin añadir más complejidad a nuestros circuitos. Está adaptado a la sensibilidad del ojo humano, por lo que no reacciona ante la luz infrarroja o ultravioleta.

Las conexiones de la plaquita breakout distribuida por Sparkfun son muy simples: el conector "VCC" puede ir conectado directamente al pin "5V" de la placa Arduino, el conector "GND" a tierra" y el conector "SIG" a cualquier entrada analógica de Arduino. Cuanto más voltaje leamos del sensor, más iluminado estará el entorno.

Ejemplo 7.6: Su programación también es muy simple. He aquí un ejemplo:

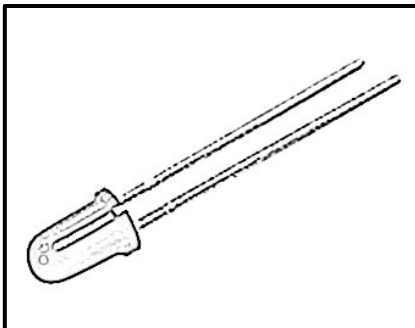
```
int pinsensor = 0; //Entrada analógica donde está conectado el sensor
void setup() {
  Serial.begin(9600);
}
void loop() {
  int lectura;
  lectura = analogRead(pinsensor);
  Serial.println(lectura); //0=muy oscuro; 1023=muy iluminado
  delay(100);
}
```

Otra plaquita breakout que incluye el mismo chip TEMT6000 es la distribuida por Freetronics bajo el nombre de "Light sensor module". Igualmente, dispone de tres conectores (VCC, GND y OUT) para comunicarse con nuestra placa Arduino y se programa exactamente igual.

Otra plaquita breakout muy parecida (con también los tres conectores y programable de la misma forma) es la llamada "AMBI Light sensor" de Modern Device, pero esta incluye otro chip, el GA1A1S201WP.

SENSORES DE LUZ INFRARROJA

Fotodiodos y fototransistores



Un fotodiodo es un dispositivo que, cuando es excitado por la luz, produce en el circuito una circulación de corriente proporcional (y medible). De esta manera, pueden hacerse servir como sensores de luz, aunque, si bien es cierto que existen fotodiodos especialmente sensibles a la luz visible, la gran mayoría lo son sobre todo a la luz infrarroja. Se pueden adquirir en cualquier distribuidor de componentes

básicos, tales como Mouser o Jameco, por poner un par de ellos. Ejemplos de dispositivos concretos que nos pueden venir bien son (el código es del fabricante) el TEFD4300F, el BPV22F, el BPV10NF o el SFH235FA.

Hay que tener en cuenta que, a pesar de tener un comportamiento en apariencia similar a los LDRs, una diferencia muy importante respecto estos (además de la sensibilidad a otras longitudes de onda) es el tiempo de respuesta a los cambios de oscuridad a iluminación, y viceversa, que en los fotodiodos es mucho menor.

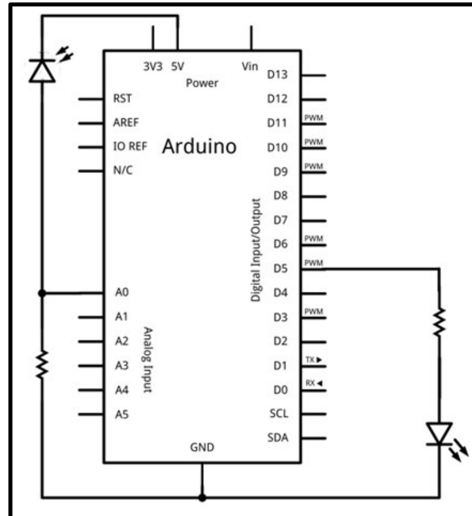
Igual que los diodos estándar, los fotodiodos poseen un ánodo y un cátodo, pero atención, para que funcione como deseamos, un fotodiodo siempre se ha de conectar al circuito en polaridad inversa. Eso sí, igual que ocurre con los diodos comunes, normalmente el ánodo es más largo que el cátodo (en caso de ser de igual longitud, el cátodo deberá estar marcado de alguna forma).

Su funcionamiento interno es el siguiente: cuando el fotodiodo está polarizado en directa, la luz que incide sobre él no tiene un efecto apreciable y por tanto el dispositivo se comporta como un diodo común. Cuando está polarizado en inversa y no le llega ninguna radiación luminosa, también se comporta como un diodo normal ya que los electrones que fluyen por el circuito no tienen energía suficiente para atravesarlo, con lo que el circuito permanece abierto. Pero en el momento en el que el fotodiodo recibe una radiación luminosa dentro de un rango de longitud de onda adecuado, los electrones reciben suficiente energía para poder "saltar" la barrera del fotodiodo en inversa y continuar su camino.

<u>Ejemplo 7.7</u>: Para probar su comportamiento, podemos utilizar un circuito como el de la página siguiente. Este circuito es idéntico al que ya vimos con los LDRs, sustituyendo estos por un fotodiodo (el cual se identifica por un nuevo símbolo que no habíamos visto hasta ahora). El valor de su divisor de tensión dependerá de la cantidad de luz (infrarroja) presente en el ambiente: resistencias mayores mejoran la sensibilidad cuando solo hay una fuente de luz y resistencias menores la mejoran cuando hay muchas (el propio sol o las lámparas son fuentes de infrarrojos); un valor de 100 KΩ puede ir bien para empezar. Fijémonos además que es el cátodo del fotodiodo (el terminal más corto, recordemos) el que se conecta a la alimentación.

El funcionamiento de este circuito es el siguiente: mientras el fotodiodo no detecte luz infrarroja, por la entrada analógica de la placa Arduino (en este caso la número 0) se medirá un voltaje de 0 V porque el circuito actuará como un circuito abierto. A medida que vaya aumentando la intensidad lumínica sobre el fotodiodo, aumentará la cantidad de electrones que lo traspasa (es decir, la intensidad de

corriente). Esto implica que, al ser la resistencia “pull-down” fija, por la Ley de Ohm el voltaje medido en el pin de entrada analógico también aumentará, hasta llegar un momento en el cual al recibir mucha luz el fotodiodo no cause apenas resistencia al paso de los electrones y por tanto la placa Arduino lea un voltaje máximo de 5 V.



Hemos añadido un LED conectado al pin de salida PWM nº 5 tal como hicimos cuando vimos los LDRs para tener una manera visible (nunca mejor dicho) de detectar la incidencia de luz infrarroja sobre el fotodiodo. Tal como se puede observar en el código utilizado (mostrado a continuación), hemos hecho depender la intensidad del brillo del LED de la cantidad de luz infrarroja detectada por el fotodiodo: cuanta más radiación infrarroja se reciba, más iluminado estará el LED.

```
int valorfotodio;    //Valor obtenido del fotodiodo
int brilloLED;       //Valor enviado al LED
void setup(void) {
       Serial.begin(9600);
}
void loop(void) {
       valorfotodio = analogRead(0);
       Serial.println(valorfotodio);
      /*El brillo del LED es proporcional a la
       cantidad de luz infrarroja recibida */
       brilloLED = map(valorfotodio, 0, 1023, 0, 255);
       analogWrite(5, brilloLED);
       delay(100); //Para que se pueda ver el nuevo brillo
}
```


Podemos jugar a aproximar el fotodiodo del ejemplo anterior a diferentes fuentes de luz infrarroja (prácticamente cualquier objeto relativamente caliente funciona como tal), pero lo más conveniente es añadir al circuito anterior una fuente de infrarrojos algo más manejable. Lo más habitual para ello es utilizar un LED emisor de infrarrojos. Simplemente conectando su ánodo al pin 5 V de la placa Arduino y su cátodo a tierra (a través de un divisor de tensión, de 220 Ω está bien) ya tendríamos una fuente constante y estable de radiación infrarroja. Si quisiéramos una fuente de emisión controlable, no tendríamos más que conectarlo como cualquier otro LED: su ánodo a un pin de salida de la placa Arduino (digital o PWM según lo que queramos) y su cátodo a tierra (a través de un divisor de tensión también, lógicamente). Sparkfun distribuye un LED emisor de infrarrojos de 850 nm con nº de producto 9469 y otro de 950 nm con nº de producto 9349. Adafruit distribuye un LED de 940 nm con nº de producto 387.

Otro tipo de sensores de luz además de los fotodiodos son los llamados fototransistores, es decir, transistores sensibles a la luz (también normalmente infrarroja). Su funcionamiento es el siguiente: al incidir luz sobre su base, en ella se genera una corriente que lleva al transistor a un estado de conducción. Por tanto, un fototransistor es igual a un transistor común con la única diferencia de que la corriente de base I_b es dependiente de la luz recibida. De hecho, existen fototransistores que pueden trabajar de las dos formas: o bien como fototransistores o bien como transistores comunes con una corriente de base I_b concreta dada.

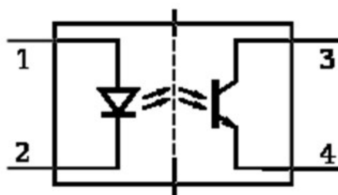
El fototransistor es mucho más sensible que el fotodiodo (por el efecto de ganancia del propio transistor), ya que las corrientes que se pueden obtener con un fotodiodo son realmente limitadas. De hecho, se puede entender un fototransistor como una combinación de fotodiodo y amplificador, por lo que, en realidad, si quisiéramos construir un fototransistor casero, bastaría con agregar a un transistor común un fotodiodo, conectando el cátodo del fotodiodo al colector del transistor y el ánodo a la base. En esta configuración, la corriente que entrega el fotodiodo (que circularía hacia la base del transistor) se amplificaría β veces.

En muchos circuitos podemos encontrar un fototransistor a poca distancia de un LED emisor de infrarrojos de una longitud de onda compatible. Esta pareja de componentes es útil para detectar la interposición entre ellos de un obstáculo (debido a la interrupción del haz de luz) y actuar por tanto como interruptores ópticos. Se pueden utilizar en multitud de aplicaciones, como por ejemplo en detectores del paso de una tarjeta de crédito (en un cajero) o de la introducción del papel (en una impresora) o como tacómetros, entre muchas otras. Un tacómetro es un dispositivo que cuenta las vueltas por minuto que realiza un obstáculo sujeto a

una rueda o aspa que gira (normalmente debido al funcionamiento de un motor); es decir, sirve para medir la velocidad de giro de un objeto.

Podemos adquirir ya de fábrica bajo un encapsulado común la pareja de componentes LED más fototransistor con el nombre genérico de “fotointerruptor”. En Sparkfun por ejemplo distribuyen uno con el código nº 9299, el cual consta de dos terminales correspondientes al ánodo y cátodo del LED, y dos terminales correspondientes al colector y emisor de un fototransistor NPN. Por lo general, querremos conectar los terminales del LED a un circuito cerrado continuamente alimentado (ánodo a fuente, cátodo a tierra), el terminal del colector del fotointerruptor a una fuente de alimentación y el terminal del emisor del fotointerruptor a una entrada digital de nuestra placa Arduino, para poder detectar así la aparición de corriente cuando se reciba iluminación. Por otro lado, tanto esta entrada de la placa Arduino como el emisor deberían estar conectados a tierra a través de la misma resistencia “pull-down”, para obtener unas lecturas más estables (un valor típico de 10 KΩ puede funcionar, pero dependiendo del circuito tal vez se necesiten valores mayores).

También podemos encontrar la pareja LED infrarrojo más fototransistor en unos componentes llamados “optoacopladores” o “optoaislador”. Su representación esquemática suele ser así:



A grandes rasgos un optoacoplador actúa como un circuito cerrado cuando llega luz desde el LED a la base del transistor y abierto cuando el LED está apagado. Su principal función es controlar y a la vez aislar dos partes de un circuito que normalmente trabajan a tensiones diferentes (tal como lo haría un transistor común, pero de una forma algo más segura). Físicamente suelen ser chips que ofrecen como mínimo cuatro patillas (igual que los fotointerruptores): dos correspondientes a los terminales del LED y dos correspondientes al colector y emisor del fototransistor (aunque pueden tener una patilla más correspondiente a la base si se permite controlar la intensidad que fluye por esta también de forma estándar). Ejemplos de optoacopladores son el 4N35 o el CNY75, fabricados por varias empresas y disponibles en Mouser, Jameco y similares.

La pareja LED-fototransistor también es útil para detectar objetos situados a pequeñas distancias de ella. Esto lo estudiaremos en el apartado correspondiente a los sensores de distancia.

Control remoto

Una utilidad práctica inmediata de una pareja emisor-receptor de infrarrojos (como un LED y un fotodiodo/fototransistor) ubicados a una cierta distancia es el envío de “mensajes” entre ellos. Es decir, ya que la luz infrarroja no es visible (y por tanto, no “molesta”), se pueden emitir pulsos de determinada duración y/o frecuencia que pueden ser recibidos y procesados a varios metros de distancia sin que “se note”. El dispositivo que los reciba deberá entonces estar programado para realizar diferentes acciones según el tipo de pulso leído. Sparkfun por ejemplo vende un “pack” con ambos componentes, con número de producto 241.

De hecho, cualquier dispositivo que funcione con un “mando a distancia” funciona de forma parecida porque en su parte frontal he de tener un sensor de infrarrojos (también llamados sensores “IR”, del inglés “infra-red”) que reciba las señales infrarrojas emitidas por el mando. Y lo que hay dentro de este es básicamente un LED que emite pulsos de luz infrarroja siguiendo un determinado patrón que señala al dispositivo la orden a realizar: existe un código de parpadeos para encender el televisor, otro para cambiar de canal, etc.

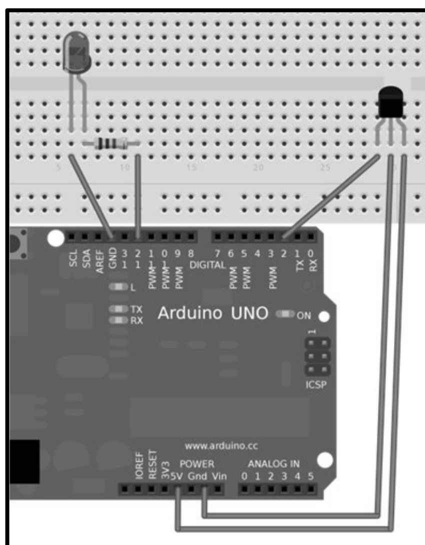
En el párrafo anterior hablamos de “sensores IR” y no de fotodiodos/fototransistores porque los primeros son algo más sofisticados. Concretamente, los sensores IR no detectan cualquier luz infrarroja, sino solo aquella que (gracias a que incorporan un filtro pasa-banda interno y un circuito demodulador) está modulada mediante una onda portadora de una frecuencia de 38 KHz ± 3 KHz. Esto básicamente quiere decir solo las señales cuya información es transportada mediante una onda de 38 KHz serán leídas. Esto es así para evitar que los sensores IR se “vuelvan locos” al recibir la luz infrarroja que existen proveniente de todos lados (sol, luz eléctrica…): de esta forma solo responden a emisiones muy concretas ya estandarizadas.

Otra diferencia con los fotodiodos/fototransistores es que los sensores IR ofrecen una respuesta binaria: si detectan una señal IR de 38 KHz el valor que se puede leer de ellos en la mayoría de los casos es LOW (0 V), y si no detectan nada, su lectura ofrece un valor HIGH (5 V). Este comportamiento es lo que se suele llamar “activo a bajo, o “low-active”.

Ejemplos de sensores IR pueden ser el TSOP32838 de Vishay (producto nº 157 de Adafruit o nº 10266 de Sparkfun) o el GP1UX311QS. Como características más destacadas tienen que su rango de sensibilidad está entre longitudes de onda de 800 nm a 1100 nm con un máximo de respuesta en 940 nm y que necesitan alrededor de 5 V y 3 mA para funcionar. Sparkfun también comercializa (con código de producto 8554) una plaquita breakout muy simple con otro sensor IR, el chip TSOP85.

El chip TSOP32838 ofrece tres patillas: teniendo de cara su dorso semiesférico, la patilla de más a la izquierda es la salida digital que ofrece el sensor, la patilla central ha de estar conectado a tierra y la patilla de más a la derecha ha de conectarse a la alimentación (de entre 2,5 V y 5,5 V).

Ejemplo 7.8: Para probar su funcionamiento, podríamos diseñar un circuito como el siguiente. El divisor de tensión para el LED puede ser de entre 200 y 1000 ohmios También se podría haber añadido un divisor de tensión de entre 100 y 900 Ω en serie a la patilla de alimentación del sensor y además, un condensador by-pass de 1 µF conectado entre esta y tierra para estabilizar el comportamiento del sensor, pero no es imprescindible.



La idea es encender durante unos instantes el LED cuando el sensor IR detecte una señal infrarroja. Pero atención, no vale cualquier señal infrarroja, sino solamente aquella modulada a 38 KHz. Por tanto, para probar este circuito, no podemos usar un LED infrarrojo cualquiera: deberemos usar algún mando de control remoto que tengamos a mano (de un televisor, un reproductor de DVD, un

computador, etc.). Una vez cargado en la placa Arduino el sketch presentado a continuación, si apuntamos con ese mando al sensor IR y pulsamos algunos de sus botones, deberíamos ver que el LED se ilumina. De esta manera, estaremos utilizando el sensor IR como si fuera un interruptor, el cual ilumina el LED mientras detecta esa señal y lo apaga cuando ya no la detecta.

El sketch necesario para que el circuito anterior funcione como deseemos es el siguiente (la salida del sensor está conectada a la entrada digital nº 2 de la placa Arduino y el LED está conectado a su salida digital nº 12):

```
int irPin=2;
int ledPin=12;
void setup() {
     pinMode(irPin,INPUT);
     pinMode(ledPin,OUTPUT);
}
void loop() {
/*Ya que la señal emitida por el sensor es normalmente HIGH, cuando
se pulsa el botón de un mando, esta cambia a LOW. Lo que hace la
función pulseIn()es pausar el sketch hasta que se detecte una señal
LOW, cuya duración en realidad no nos interesa pero lógicamente
siempre será mayor de cero. Por tanto, si se cumple la condición del
if significa que se ha pulsado un botón de un mando */
        if(pulseIn(irPin,LOW) > 0) {
/*Es necesario esperarse un tiempo determinado (que dependerá del
modelo concreto de mando a distancia) tras la detección de la primera
señal LOW debido a que cada pulsación de botón produce múltiples
oscilaciones entre valores HIGH y LOW. Aunque físicamente no tiene
nada que ver, podemos entender esta espera como si fuera una manera
de evitar un “bounce” (fenómeno estudiado cuando tratamos los
pulsadores). Una vez transcurrido ese tiempo de espera, la señal del
sensor debería haber vuelto a su estado de reposo (valor HIGH).*/
             delay(100);
/*Mantenemos encendido el LED durante unos cuantos milisegundos.
Durante este tiempo el sketch no podrá detectar otras pulsaciones
provenientes del control remoto. También podríamos haber enviado un
mensaje al “Serial monitor” notificando la pulsación. */
             digitalWrite(ledPin,HIGH);
             delay(200);
             digitalWrite(ledPin,LOW);
        }
}
```

Sin embargo, más allá de aquí no podremos hacer gran cosa con nuestro Arduino si no conocemos los patrones concretos de pulsos IR emitidos por el mando a distancia utilizado. Es decir, para evitar que un mando a distancia Sony pueda cambiar los canales de un televisor Philips (por ejemplo), cada marca utiliza una codificación distinta para sus señales, aunque todas estén moduladas a 38 KHz. Es decir, cada marca emite diferentes duraciones, secuencias y combinaciones de señales HIGH y LOW, para evitar así posibles interferencias. Por lo tanto, si queremos controlar una placa Arduino mediante un mando a distancia de algún aparato que tengamos a mano de una determinada marca, deberíamos conocer primero el protocolo utilizado por este para que nuestra placa lo procese adecuadamente. Y también a la inversa: si queremos controlar remotamente un electrodoméstico de una determinada marca mediante una placa Arduino (que haría en este caso de mando a distancia) deberíamos conocer el protocolo de comunicación reconocido por ese electrodoméstico. Desgraciadamente, hay casi tantos patrones de pulsos IR como aparatos, pero tenemos varias soluciones para ello.

La primera sería adquirir una pareja de mando a distancia más sensor IR que fueran compatibles de fábrica, sin tenernos que preocupar de su protocolo interno de comunicación. En este sentido, podemos adquirir el “IR Kit” de DFRobot (producto nº DFR0107), que incluye un mando a distancia y un sensor IR compatible (con nº de producto DFR0094 si se adquiere por separado). Este sensor viene dentro de una plaquita breakout con tres pines (5V, GND y señal digital) que podemos conectar muy fácilmente a nuestra placa Arduino o a una breadboard. Lo interesante es que este kit viene acompañado de una librería descargable de la web del producto, que permite a nuestra placa Arduino interpretar de una forma adecuada los datos recibidos por el pin de señal digital del sensor IR.

Otra alternativa similar es la adquisición, en Adafruit, de la pareja formada por el mando a distancia con nº de producto 389 (que sí se sabe que utiliza el patrón de señales NEC) y el sensor IR TSOP38238, utilizado en el circuito de ejemplo anterior. Si se utiliza con el mando a distancia indicado, este sensor puede ser controlado mediante una librería propia de Adafruit, la “Adafruit NEC Remote Control Library”, descargable de https://github.com/adafruit/Adafruit-NEC-remote-control-library

Sparkfun por su parte distribuye un kit completo con nº de producto 10783, el cual incluye un mando a distancia (con nº de producto 10280 si se adquiere por separado), un par de sensores IR también de modelo TSOP38238 y además un par de LEDs IR también. Para poder procesar correctamente las señales emitidas por el mando a distancia distribuido en este kit (y solo ese mando en particular), no existe

una librería oficial, pero en la página del producto se nos ofrece un código Arduino de ejemplo donde se hace uso de una función propia muy sencilla de utilizar llamada *getIRkey()*, la cual devuelve el código numérico correspondiente al botón pulsado en ese momento (o 0 si no hay pulsado ninguno). Por otro lado, junto con el mismo mando a distancia, también se puede utilizar la plaquita breakout con nº de producto 8554 mencionada unos párrafos más arriba; en este caso, sí se puede utilizar una librería, disponible en https://github.com/konstantint/ArduinoSparkfunIRReceiver.

Otro enfoque diferente a las soluciones anteriores para conseguir el control remoto de nuestro proyecto es no utilizar ningún mando a distancia, sino construírselo uno mismo. Es más fácil de lo que puede parecer a priori, ya que para emitir una señal infrarroja modulada a 38 KHz lo único que se necesita es un simple LEDs infrarrojo. Eso sí, es necesario controlarlo de tal forma que pueda encenderse y apagarse a 38 KHz el número de veces que sea necesario. Es decir, si queremos emitir con un LED infrarrojo una señal HIGH que dure (por ejemplo) un segundo y el LED simplemente permanece encendido durante ese tiempo y ya está, el sensor IR no detectará nada porque la señal no está modulada. Para poder modular esa señal HIGH el LED deberá encenderse y apagarse 38000 veces durante ese segundo. De esta manera, el sensor IR reconocerá ese patrón de frecuencia e interpretará que le ha llegado un pulso. Si quisiéramos emitir una señal LOW, con mantener el LED apagado ya valdría.

Pensando en esta aplicación, Sparkfun incluso comercializa una plaquita breakout (con código 10732) que incluye un LED infrarrojo y un transistor, de forma que la luz emitida tenga una mayor intensidad y sea más fácilmente captarla a mayores distancias. Tan solo es necesario enviar la señal modulada a la base del transistor (correspondiente al conector de la plaquita etiquetado como "CTL"), además de alimentar dicha plaquita y conectarla a tierra.

Ejemplo 7.9: A continuación, se muestra un código de ejemplo de la emisión de un patrón ficticio de pulsos modulados a 38 KHz, repetido infinitamente cada segundo. Este patrón está formado por pulsos de diferente duración, separados por intervalos de tiempo también de distinta duración. Así se pretende simular el patrón de un producto comercial cualquiera, en el cual los pulsos suelen ser de distinta longitud y aparecer en intervalos diferentes. Supondremos que el LED está conectado directamente al pin de salida digital nº 8 de la placa Arduino (y a tierra, a través de un divisor de tensión, como siempre).

```
void setup(){
    pinMode(8, OUTPUT);
}
void loop(){
    pulseIR(2080);//Emito un pulso modulado durante 2080 µs
    delay(27);    //No emito nada durante 27 milisegundos
    pulseIR(440); //Emito un segundo pulso modulado durante 440 µs
    delayMicroseconds(1500);//No emito nada durante 1500 µs
    pulseIR(460);
    delayMicroseconds(3440);
    pulseIR(480);
    delay(1000);
}
/*Esta función envía un pulso modulado a 38KHz
de una duración determinada por su parámetro.*/
void pulseIR(long microsecs) {
    cli();
    /*Voy pasando los periodos que "caben"
    en el intervalo de tiempo especificado */
    while (microsecs > 0) {
/*Una frecuencia de 38KHz equivale a un periodo de 26 microsegundos:
durante 13 microseconds (semiperiodo) la señal ha de ser HIGH y
durante 13 microseconds será LOW */
        digitalWrite(8,HIGH);
/*Es conocido que la ejecución de la función digitalWrite tarda sobre
3 microsegundos, por lo que tras ella espero 10 microsegundos más */
        delayMicroseconds(10);
        digitalWrite(8, LOW);         //Se tarda también 3 µs
        delayMicroseconds(10);        //Me espero 10 µs más
        microsecs = microsecs - 26;   //Ya se ha cumplido un periodo
    }
    sei();
}
```

En el código anterior aparecen dos funciones del lenguaje Arduino que no habíamos visto hasta ahora: **cli()** y **sei()**. La función *cli()* sirve para desactivar ciertas tareas que la placa Arduino siempre está realizando constantemente en segundo plano (tales como escuchar posibles datos recibidos por el puerto serie, llevar la cuenta del tiempo, etc.). Cuando se están tratando señales de alta velocidad, es muy recomendable mantener la placa Arduino lo más inactiva posible para poder realizar un seguimiento preciso y limpio. La segunda sirve para volver activar estas tareas.

Ejemplo 7.10: No serviría de nada tener un emisor de pulsos modulados si no tenemos un sensor que los reconozca. Para ello, utilizaremos el sensor IR TSOP38238 (con su salida conectada al pin nº 2 de la placa Arduino) y el siguiente código, el cual mostrará por el Serial monitor la duración de los pulsos modulados recibidos (valores HIGH), y el tiempo existentes entre ellos (en realidad, la duración de los pulsos LOW). Con este código, de hecho, podremos conocer los patrones de cualquier mando a distancia comercial y por tanto, podremos construirnos un clon:

```
//Pin de la placa Arduino donde está conectada la salida del sensor
const byte irPin=2;
//Duración máxima que reconoceremos para un pulso (¡65 ms es mucho!)
const int MAXPULSO = 65000;
/*Resolución temporal que utilizaremos. Cuanto menor sea su valor,
más precisa será la lectura de la señal, pero si es demasiado pequeño
puede haber problemas de sincronización. El valor de 20 suele ser el
adecuado en la mayoría de los casos*/
const byte RESOLUCION = 20;
/*Guardaremos 100 parejas formadas por un pulso (valor HIGH) y su
valor LOW siguiente. Para ello hacemos uso de un array bidimensional:
el valor HIGH de la primera pareja se guardará en el elemento
pulsos[0][0], el valor LOW de la primera pareja se guardará en el
elemento pulsos[0][1], el valor HIGH de la segunda pareja se guardará
en el elemento pulsos[1][0], el valor LOW de la segunda pareja se
guardará en el elemento pulsos[1][1], el valor HIGH de la tercera
pareja se guardará en el elemento pulsos[2][0], y así. */
word pulsos[100][2];
//Índice del pulso que estamos guardando en ese momento en el array
byte pulsoactual = 0;
setup(){
	Serial.begin(9600);
}
loop(){
	word contadortiempohigh =0;
	word contadortiempolow  =0;
/*La siguiente línea es igual a "while(digitalRead(irPin)){", pero no
la escribimos así porque la función digitalRead() es demasiado lenta
para poder leer las rápidas variaciones de la señal modulada. Por
tanto, hemos recurrido a código C optimizado para la plataforma AVR,
el cual nos da un acceso más directo al hardware de la placa. No
profundizaremos más en ello. */
	//Mientras se recibe una señal HIGH
	while ((PIND & _BV(irPin))) {
		contadortiempohigh++;
```

```
            //Cuento unos cuantos microsegundos más
            delayMicroseconds(RESOLUCION);
/*Si el pulso es demasiado largo significa que ya se acabó el patrón,
por lo que lo imprimimos, nos preparamos para rellenar otra vez el
array y, mediante la función return, volvemos al inicio de la función
loop()*/
      if ((contadortiempohigh >= MAXPULSO) && (pulsoactual != 0)) {
                  printpulsos();
                  pulsoactual=0;
                  return;
            }
      }
      /*Ya se ha dejado de recibir un pulso HIGH,
       por lo que guardamos el tiempo que ha durado.*/
      pulsos[pulsoactual][0] = contadortiempohigh;
      //Seguimos leyendo el siguiente pulso de la señal por el pin 2
      //Mientras se recibe una señal LOW
      while (!(PIND & _BV(irPin))) {
            contadortiempolow++;
            delayMicroseconds(RESOLUCION);
      if ((contadortiempolow >= MAXPULSO) && (pulsoactual != 0)) {
                  printpulsos();
                  pulsoactual=0;
                  return;
            }
      }
      pulsos[pulsoactual][1] = contadortiempolow;
//Hemos leído una pareja de valores HIGH-LOW con éxito. Continuamos
      pulsoactual++;
}
void printpulsos() {
      byte i;
      Serial.println("OFF \tON");
      for (i = 0; i < pulsoactual; i++) {
            Serial.print(pulsos[i][0] * RESOLUCION);
            Serial.print(" usec, ");
            Serial.print(pulsos[i][1] * RESOLUCION);
            Serial.println(" usec");
      }
}
```

De todas formas, existe una forma mucho más sencilla de poder utilizar señales moduladas: descargar e instalar la librería “Arduino IR”, disponible en la página https://github.com/shirriff/Arduino-IRremote . Con ella no tendremos ni que

adquirir ningún nuevo mando específico (por lo que podremos reciclar alguno que tengamos a mano de un aparato ya inútil) ni tampoco tendremos que investigar su patrón particular para poderlo controlar. Esta librería permite que nuestra placa Arduino pueda tanto enviar como recibir códigos de control remoto en múltiples patrones ya incluidos "de fábrica" (soporta los protocolos NEC, Sony SIRC, Philips RC5 y RC6, y muchos más) e incluso tiene un mecanismo para añadirle más protocolos si es necesario. Con ella, nuestra placa Arduino incluso podría funcionar como un reenviador de códigos, recibiéndolos de un mando a distancia comercial y retransmitiéndolos a otro lugar. Veamos algún ejemplo.

Ejemplo 7.11: Suponiendo que tenemos conectado el sensor IR TSOP38238 (u otro similar) a un pin de entrada digital cualquiera de Arduino, el siguiente código muestra por el "Serial monitor" los comandos recibidos de un mando a distancia cualquiera.

```
#include <IRremote.h>
/*Pin de entrada digital de la placa Arduino donde
 hemos colocado la patilla de señal del receptor */
int pinreceptor = 11;
//Creo un objeto llamado "irrecv" de tipo IRrecv
IRrecv irrecv(pinreceptor);
//Declaro una variable de un tipo especial, "decode_results".
decode_results resultados;
void setup(){
  Serial.begin(9600);
  irrecv.enableIRIn(); //Inicio el receptor
}
void loop() {
/*Miro si se ha detectado algún patrón IR modulado. Si es así, lo leo
y lo guardo enteramente en la variable especial "resultados", en
forma de número hexadecimal*/
  if (irrecv.decode(&resultados) !=0) {
        /*Primero miro qué tipo de patrón comercial es,
        si es que es de alguno reconocido por la librería */
        if (results.decode_type == NEC) {
                     Serial.print("NEC: ");
        } else if (results.decode_type == SONY) {
                     Serial.print("SONY: ");
        } else if (results.decode_type == RC5) {
                     Serial.print("RC5: ");
        } else if (results.decode_type == RC6) {
                     Serial.print("RC6: ");
          } else if (results.decode_type == UNKNOWN) {
                       Serial.print("Desconocido: ");
```

```
          }
        /*Y seguidamente muestro el patrón recibido
        (en formato hexadecimal) por el canal serie */
        Serial.println(resultados.value, HEX);
        /*Una vez el patrón ha sido decodificado, reactivo otra vez la
        escucha para poder detectar el siguiente posible patrón */
        irrecv.resume();
  }
/*Aquí se pueden hacer otras cosas mientras
 se espera a recibir un comando IR*/
}
```

Tal como se expresa en los comentarios del código anterior, la función *decode()* no es bloqueante; esto quiere decir que el sketch puede realizar otras operaciones mientras se está esperando la detección de un patrón nuevo.

Cada botón de un mando a distancia está asociado a un patrón particular (generalmente de 12 a 32 pulsos) identificable mediante un número hexadecimal. Si el botón se mantiene pulsado, este código usualmente es repetido constantemente, aunque existen mandos que solo envían el código una sola vez y utilizan otros métodos para detectar el fin de una pulsación (como marcas de final de pulsación o códigos especiales que funcionan como contadores, etc.).

Ejemplo 7.12A: Una vez conocido el patrón de un mando a distancia, podemos desarrollar sketches que hagan reaccionar a nuestra placa Arduino según el botón que haya sido pulsado. Por ejemplo, en el siguiente sketch podemos observar por el "Serial monitor" el botón pulsado de un mando a distancia de un televisor, concretamente de la marca Sony.

```
#include <IRremote.h>
int pinreceptor = 11;
IRrecv irrecv(pinreceptor);
decode_results resultados;
void setup(){
  Serial.begin(9600);
  irrecv.enableIRIn();
}
void loop(){
   int i;
   if (irrecv.decode(&resultados)!=0) {
```

```
          accion();
          /*Los mandos Sony envían 3 veces el mismo patrón repetido.
          Este "for" sirve para ignorar el 2° y 3° patrón.*/
          for (i=0; i<2; i++) {
               //Recibo el siguiente patrón y no hago nada
                irrecv.resume();
          }
   }
}
void accion() {
  switch(results.value)  {
    case 0x37EE: Serial.println("Favoritos"); break;
    case 0xA90: Serial.println("Encendido/Apagado"); break;
    case 0x290: Serial.println("Mute"); break;
    case 0x10:  Serial.println("1"); break;
    case 0x810: Serial.println("2"); break;
    case 0x410: Serial.println("3"); break;
    case 0xC10: Serial.println("4"); break;
    case 0x210: Serial.println("5"); break;
    case 0xA10: Serial.println("6"); break;
    case 0x610: Serial.println("7"); break;
    case 0xE10: Serial.println("8"); break;
    case 0x110: Serial.println("9"); break;
    case 0x910: Serial.println("0"); break;
    case 0x490: Serial.println("Aumentar volumen"); break;
    case 0xC90: Serial.println("Disminuir volumen"); break;
    case 0x90:  Serial.println("Aumentar canal"); break;
    case 0x890: Serial.println("Disminuir canal"); break;
    default: Serial.println("Otro botón");
  }
  delay(500);
}
```

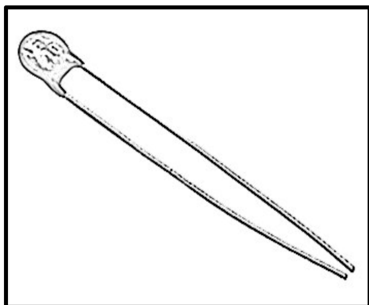
Ejemplo 7.12B: Para hacer lo contrario, es decir, para enviar un patrón comercial determinado mediante un LED infrarrojo conectado a nuestra placa Arduino, sería tan sencillo como ejecutar un sketch similar al siguiente. Mediante este código de ejemplo, nuestra placa Arduino encenderá (o apagará) un televisor que utilice el protocolo de Sony cada vez que reciba cualquier dato a través de su canal serie. Un detalle muy importante a tener en cuenta es que para que el código siguiente funcione, el LED infrarrojo ha de estar conectado obligatoriamente al pin digital de salida nº 3 de nuestra placa Arduino.

```
#include <IRremote.h>
IRsend irsend;
void setup(){
  Serial.begin(9600);
}
void loop() {
  if (Serial.read() != -1) {
  /*El patrón se ha de enviar tres veces porque
  el protocolo Sony lo establece así */
       for (int i = 0; i < 3; i++) {
/*La función sendSony() envía un patrón especificado como primer
parámetro perteneciente a ese protocolo (en este caso, el de
encendido/apagado). El segundo parámetro indica el número de bits que
componen ese patrón. Este número es simplemente el número de dígitos
hexadecimales de los que se compone el patrón multiplicado por 4.*/
              irsend.sendSony(0xA90, 12);
              delay(100);
         }
    }
}
```

Existen otras funciones predefinidas diferentes de *sendSony()* para poder enviar otro tipo de protocolos comerciales comunes, como por ejemplo *sendNEC() , sendRC5() , sendRC6(), sendSharp()* o *sendPanasonic() .* Todas ellas funcionan con los dos mismos parámetros. Además, se puede usar la función *sendRaw()* para enviar un patrón que no venga por defecto codificada dentro de la librería. Esta función tiene tres parámetros: el primero es un array de valores word que forman en conjunto el patrón completo, el segundo es el tamaño de ese array y el tercero es la frecuencia de modulación en KHz de la señal a enviar (que para nuestros proyectos siempre será 38). Lo más interesante de esta función es que podemos utilizarla para simular cualquier protocolo comercial a pesar de que este no venga por defecto codificado dentro de la librería. Esto lo podemos hacer eligiendo previamente de entre los sketches de ejemplo que vienen con la librería uno llamado “IRrecvDump.ino” y ejecutándolo con nuestro Arduino conectado a un sensor IR. Haciendo esto podremos observar en el “Serial monitor” precisamente los valores word a incluir en el array (el primer parámetro de *sendRaw()*), y también, entre paréntesis, el tamaño de dicho array (el segundo parámetro de *sendRaw()*).

SENSORES DE TEMPERATURA

Termistores



Un termistor es un resistor que cambia su resistencia con la temperatura. Técnicamente, todos los resistores son termistores ya que su resistencia siempre cambia ligeramente con la temperatura, pero este cambio es usualmente muy pequeño y difícil de medir. Los termistores están fabricados de manera que su resistencia cambia drásticamente, de tal manera que pueden cambiar 100 ohmios o más por grado centígrado.

Hay dos tipos de termistores, los llamados NTC (del inglés "negative temperature coefficient") y los PTC (de "positive temperature coefficient"). En los primeros, a medida que aumenta la temperatura, decrece su resistencia; en los segundos, a medida que aumenta la temperatura, aumenta su resistencia. En nuestros proyectos normalmente usaremos NTCs para medir temperatura; los PTCs se suelen usar más dentro de fusibles reseteables (donde si la temperatura crece, incrementan su resistencia para "ahogar" la corriente y proteger así de un posible sobrecalentamiento a los circuitos).

Los termistores son mucho más baratos que otros tipos de sensores de temperatura; además, son resistentes al agua (son solo resistores al fin y al cabo) y trabajan a cualquier voltaje. Son difíciles de estropear debido a su sencillez y son increíblemente precisos en las medidas. Por ejemplo, un termistor de 10 KΩ (valor nominal, tomado a 25 °C como referencia estándar) puede medir temperatura con un margen de error de ±0,25 °C (suponiendo que el conversor analógico-digital sea lo suficientemente preciso también). No obstante, no suelen soportar temperaturas más allá de los 100 y poco grados, y su constante de tiempo (es decir, los segundos que necesita el termistor para reducir un 63% la diferencia entre su temperatura inicial y la final) es normalmente de más de diez segundos.

Para medir la resistencia de un termistor se puede usar un multímetro, como cualquier otra resistencia. El valor que obtengamos dependerá de la temperatura del lugar donde estemos.

Si lo que queremos es medir la temperatura propiamente dicha con una placa Arduino, primero debemos medir con ella la resistencia del termistor, y a partir de

ella deducir la temperatura correspondiente. Pero nuestra placa Arduino no tiene un medidor de resistencias incorporado, por lo que, al igual ya hicimos con los fotorresistores, tendremos que utilizar las entradas analógicas de la placa para detectar variaciones de voltaje y deducir a partir de estas el valor buscado de la resistencia actual. Así pues, el esquema de conexiones es idéntico al que ya usamos con los fotorresistores (y fotodiodos): debemos conectar un terminal del termistor a la alimentación (por ejemplo, el pin 5V de la placa Arduino) y el otro conectarlo en serie a un terminal de una resistencia de valor fijo (que hará de resistencia "pull-down"); el otro terminal de esta resistencia "pull-down" ha de conectarse a tierra. Además, deberemos conectar una entrada analógica de nuestra placa Arduino a un punto intermedio entre ambas resistencias para obtener una lectura de la caída de potencial entre ese punto y tierra.

Mediante el circuito acabado de describir, cuando detectemos que ese voltaje medido está aumentando, podremos deducir, por pura Ley de Ohm, que la resistencia del termistor NTC está disminuyendo y, por tanto, que la temperatura está aumentando (y viceversa: si el voltaje disminuye, la resistencia aumenta y la temperatura disminuye también). La relación exacta entre el voltaje medido y la resistencia del termistor se puede calcular de la misma manera que ya vimos cuando estudiamos los fotorresistores, mediante la fórmula $V_{med}=(R_{pull}/(R_{pull}+R_{termistor}))\cdot V_{fuente}$.

Ya sabemos también que, en realidad, V_{med} no es el voltaje con el que trabajamos en nuestra placa Arduino, porque esta utiliza siempre un conversor analógico-digital con el que realiza un "mapeo". Este mapeo convierte los valores analógicos leídos (los cuales pueden ir oscilando entre 0 V y 5 V si suponemos que el voltaje proporcionado por la fuente son 5 V) a valores digitales (que van entre 0 y 1023). Estos valores digitales son los que la placa Arduino entiende en realidad y con los que trabajaremos en nuestros sketches. La conversión de valores analógicos a digitales se puede expresar por la misma regla de proporcionalidad que vimos con los fotorresistores: $V_{convertido} = V_{med} \cdot 1023/5$. A partir de aquí, tal como hicimos con ellos, si sustituimos esta expresión en la fórmula del párrafo anterior, y despejamos de allí $R_{termistor}$ llegamos a la expresión siguiente: $R_{termistor}=(R_{pull}\cdot 1023/V_{convertido}) - R_{pull}$, la cual nos permite conocer por fin cuál es el valor actual de la resistencia del termistor a partir del voltaje digitalizado obtenido por la placa Arduino. La expresión anterior demuestra, tal como ya sabíamos, que $R_{termistor}$ es inversamente proporcional a $V_{convertido}$.

Pero ¿qué valor ha de tener R_{pull}? El valor más habitualmente empleado para la resistencia "pull-down" que acompaña a nuestro termistor es de 10 KΩ, aunque a veces también se usa 1 KΩ. Ambos son valores ampliamente utilizados, pero si queremos tener un control mayor de la sensibilidad del termistor, tendríamos que

elegir su valor con algo más de criterio. Para ello, el procedimiento sería sustituir en la misma fórmula ya conocida $V_{med}=(R_{pull}/(R_{pull}+R_{termistor}))\cdot V_{fuente}$ diferentes valores numéricos concretos de $R_{termistor}$ correspondientes a temperaturas conocidas (esto se puede consultar en la tabla de equivalencia del datasheet) y elegir un valor determinado de R_{pull} : observando qué valores de V_{med} vamos obteniendo podremos elegir el valor R_{pull} que nos permita tener un rango más amplio de V_{med} sin saturarlo.

El paso natural siguiente, una vez conocido el valor de la resistencia del termistor, sería averiguar a qué temperatura se corresponde. La manera más sencilla de conseguir esto es consultar la tabla de equivalencia que siempre viene en el datasheet, donde para unos determinados valores de $R_{termistor}$ se especifica ya directamente la temperatura correspondiente.

Este sistema nos irá bien si lo único que queremos es diseñar un circuito que realice comparaciones rápidas del tipo “si la temperatura es inferior a tal haz esto y si es superior a tal haz lo otro”. Si lo que queremos es, sin embargo, obtener los valores de temperatura reales y precisos, tenemos la suerte de disponer de una ecuación matemática que hace esto con muy buena aproximación (de hecho, logra errores de tan solo ±0,02 ^{o}C en un rango de 100 ^{o}C). Se trata de la ecuación de Steinhart-Hart: $\frac{1}{T} = A + B \cdot \ln(R) + \ C \cdot (\ln(R))^3$, donde R es el valor en ohmios del termistor en un momento dado, T es la temperatura medida en grados Kelvin (un grado Kelvin es igual a uno centígrado + 273,15) y A, B y C son coeficientes que son diferentes según el tipo y modelo del termistor (y que son válidos solamente para un determinado rango de temperaturas, especificado en el datasheet).

No obstante, como esta fórmula es algo compleja y requiere el conocimiento del valor de varios coeficientes que puede que no conozcamos, en general podemos usar una ecuación simplificada: $\frac{1}{T} = \frac{1}{T_0} + \frac{1}{B} \cdot \ln\left(\frac{R}{R_0}\right)$, donde T_0 es la llamada temperatura nominal (casi siempre 25 °C = 298,15 K), R_0 es la resistencia del termistor a esa temperatura (la llamada “resistencia nominal”, dato disponible en el datasheet) y B es el único coeficiente que necesitamos saber, el cual se puede consultar en el datasheet siempre.

Ejemplo 7.13: Una vez sabida toda la teoría previa, ya podemos implementar el circuito de prueba y empezar a escribir código. Tal como hemos dicho, conectaremos un terminal del termistor a 5 V y el otro lo conectaremos a un terminal de la resistencia “pull-down”. El otro terminal de esta última lo conectaremos a tierra y finalmente usaremos un “tercer cable” para conectar un punto central entre ambos dispositivos a un pin analógico de la placa Arduino (por ejemplo, el número 0).

El código mostrado a continuación muestra por el canal serie los valores de $R_{termistor}$, ya calculados a partir de la lectura obtenida de la entrada analógica. No se realiza ningún cálculo de temperatura.

```
const int Rpull=10000;
void setup(void) {
       Serial.begin(9600);
}
void loop(void) {
//El tipo de datos es importante para los cálculos posteriores
       float lectura;
       lectura = analogRead(0);
       Serial.print("Lectura analógica directa ");
       Serial.println(lectura);
      /*Convierto “lectura”, que es un valor de voltaje,
        en resistencia, según la fórmula ya vista.*/
       lectura = (Rpull * 1023/lectura)  - Rpull);
       Serial.print("Resistencia del termistor ");
       Serial.println(lectura);
       delay(1000); //Para la estabilidad en las lecturas
}
```

El siguiente trozo de código realiza todos los cálculos necesarios para obtener la lectura en grados centígrados a partir de la ecuación de Steinhart-Hart simplificada. Estas líneas se tendrían que añadir al código anterior, justo antes de su última línea *delay(1000);*. Para que funcione además hay que declarar e inicializar una serie de variables extra: “termistorNominal” con valor 10000, “temperaturaNominal” con valor 25 y “coeficienteB” con el valor concreto de nuestro termistor (en nuestro caso, 3950).

```
float steinhart;
steinhart = media / termistorNominal;           // (R/Ro)
steinhart = log(steinhart);                     // ln(R/Ro)
steinhart /= coeficienteB;                      // 1/B * ln(R/Ro)
steinhart += 1.0/(temperaturaNominal + 273.15); // + (1/To)
steinhart = 1.0 / steinhart;                    // Se invierte
steinhart -= 273.15;                            // Se convierte a ℃
Serial.print("Temperatura: ");
Serial.print(steinhart);
Serial.println(" *C");
```

El valor devuelto tras ejecutar las líneas anteriores es una cifra con dos dígitos decimales. ¿Esto quiere decir que se tiene una precisión de 0,01 °C? ¡No! El termistor tiene errores y el conversor analógico-digital también. Afortunadamente, podemos

estimar el error de nuestra medida a partir del error nominal del termistor (dato consultable en el datasheet con el nombre de “tolerancia”). Supongamos que a la temperatura nominal (25 °C) tenemos un termistor de 10000 Ω con un error (una tolerancia) del 1%, es decir, 100 Ω. Esto implica que para esa temperatura se podrían leer valores desde 9900 Ω hasta 10100 Ω. De la ecuación de Steinhart-Hart simplificada se puede deducir que, si ese termistor tiene un coeficiente B de por ejemplo 3950, una diferencia de 450 Ω arriba y abajo representaría 1 °C de diferencia, así que un error de 100 Ω arriba y abajo (su tolerancia) significa un error de alrededor ±0,25 °C.

Desgraciadamente, aunque existen termistores con tolerancias incluso del 0,1% (los cuales permiten reducir el error hasta ±0,03 °C), la inexactitud en las medidas en realidad siempre es mayor, porque siempre existe como mínimo otra fuente de error que no podemos olvidar: el producido por el conversor analógico-digital. En el caso del conversor presente en la placa Arduino (que tiene una resolución de 10 bits), por cada bit erróneo la resistencia medida puede variar hasta 50 ohmios de la real. Esto representa un error de ±0,1 °C más a sumar a los errores anteriores. En general, por tanto, es recomendable asumir una precisión no mejor que ±0,5 °C.

También hay que tener en cuenta que el termistor sufre un autocalentamiento consecuencia de la potencia disipada, por lo que alcanza una temperatura por encima de la del ambiente, que es a su vez la temperatura que detecta. Esto obliga a limitar el valor de la tensión (o de la corriente, recordemos que P = V·I) con la que se alimenta. En este sentido, la resistencia “pull-down” es de mucha ayuda, porque funciona como un divisor de tensión.

Ejemplo 7.14: En general, cuando se hacen lecturas de valores analógicos, hay dos trucos y que sirven para mejorar la precisión de los resultados. Uno es utilizar el pin 3,3 V como voltaje de referencia analógico (para aumentar la precisión de la conversión analógico-digital) y el otro es tomar un conjunto de lecturas y obtener su media como el resultado a usar (para evitar fluctuaciones o ruido en las medidas).

Para conseguir el primer truco, simplemente conectaremos el pin “3V3” de la placa a su pin AREF (además de añadir una línea con *analogRead()* específica a nuestro código). Para conseguir el segundo truco, modificaremos el código para incluir bucles que recojan varias medidas (cinco están bien) y realicen las medias pertinentes. Así:

```
/*Dependiendo del número de muestras, la media
tarda más (por el bucle) pero es más suave*/
const int numMuestras=5;
const int Rpull=10000;
```

```
int muestras[numMuestras];
void setup(void) {
       Serial.begin(9600);
       analogReference(EXTERNAL);
}
void loop(void) {
       byte i;
       float media;
       for (i=0; i< numMuestras; i++) {
          muestras[i] = analogRead(0);
          delay(10);  //Para la estabilidad entre lecturas
       }
       //Calculo la media
       media = 0;
       for (i=0; i< numMuestras; i++) {
          media = media + muestras[i];
       }
       media = media/numMuestras;
       Serial.print("Lectura analógica media: ");
       Serial.println(media);
       //Convierto "media" a resistencia, según la fórmula
       media= (Rpull * 1023/media)  - Rpull);
       Serial.print("Resistencia del termistor ");
       Serial.println(media);
       delay(1000);
}
```

El chip analógico TMP36

Este chip utiliza una tecnología de estado sólido para medir la temperatura: a medida que la temperatura crece, la caída de potencial entre la Base y el Emisor de un transistor incrementa también una cantidad conocida. Amplificando este cambio de voltaje, se genera una señal analógica que es directamente proporcional a la temperatura. Este tipo de sensores son precisos, no se desgastan, no necesitan calibración, pueden trabajar bajo condiciones climáticas diversas, son bastante baratos y fáciles de usar. En Adafruit se distribuye con código de producto 165 y en Sparkfun con código 10988.

Otros detalles técnicos: su rango de temperatura va desde los -4 0°C hasta los 125 °C y su rango de voltaje de salida

empieza desde 0,1 V (a los -40 °C) y aumenta 10 mV por cada grado centígrado hasta llegar a los 1,75 V (a los 125 °C). Por otro lado, para que su circuitería interna funcione, necesita estar alimentado por una fuente de entre 2,7 V y 5,5 V y 0,05 mA.

Sus conexiones son sencillas: si se tiene enfrente su parte plana (tal como se muestra en la figura), el pin de más a la izquierda (nº 1) ha de conectarse a la alimentación y el de más a la derecha (nº 3) a tierra. El pin del medio (nº 2) es el que sirve para obtener un voltaje analógico linealmente proporcional a la temperatura (e independiente del voltaje proporcionado por la fuente de alimentación). Por tanto, este pin nº 2 se tendrá que conectar a un pin analógico de entrada de nuestra placa Arduino.

Luego, para convertir el voltaje leído en temperatura, simplemente hay que utilizar la siguiente fórmula extraída del datasheet: T = (V - 0,5)*100 donde T se mediría en grados centígrados y V en voltios. Así, por ejemplo, si se mide un voltaje de 1V, la temperatura sería (1V - 0,5)*100 = 50 °C. De la fórmula anterior podemos deducir un par de cosas: que la resolución es de 10 mV por grado (tal como ya hemos comentado) y que a 0 °C existe un voltaje de 500 mV. Esta característica permite que el sensor pueda devolver lecturas de temperaturas bajo cero sin que tengamos que preocuparnos por manejar valores de voltaje negativos.

Si se desea probar este chip antes de incorporarlo a nuestros proyectos, podemos usar un multímetro en modo medición de voltaje DC. Tendremos que alimentar no obstante al chip mientras dure la medición. Para ello, podemos usar dos pilas AA (3 V entra perfectamente dentro del rango de voltaje admitido por el chip), de tal manera que conectemos su borne positivo y negativo a los pines correspondientes. El multímetro deberemos conectarlo al pin 3 (tierra) y al pin 2 para realizar la medición. En una habitación a 25 °C, el voltaje debería de ser de alrededor de 0,75 V. Se puede jugar a apretar los dedos sobre el encapsulado (para calentarlo) o ponerlo en contacto con hielo (para enfriarlo) y ver los cambios de valores medidos.

<u>Ejemplo 7.15</u>: A continuación mostramos un código Arduino que muestra por el canal serie la temperatura (ya calculada) del ambiente. El circuito tan solo consta de este componente conectado a los pines-hembra adecuados de Arduino (alimentación, tierra y pin de entrada analógico, que supondremos el nº 0), y ya está. El voltaje de alimentación podría ser 5 V como siempre o bien los 3,3 V proporcionados por el pin “3V3”: los resultados son independientes de eso. Lo que sí se puede hacer es utilizar el pin “3V3” también como voltaje de referencia conectándolo al pin “AREF” e indicándolo en el código: con esto se mejoraría la precisión de los resultados.

```
void setup(){ Serial.begin(9600); }
void loop() {
	int lectura;
	float voltaje;
	float temperaturaC;
	lectura = analogRead(0);
/*Convierto la lectura (0-1023) en voltaje (0-5). Si el chip se ha
alimentado con el pin “3V3”, en la fórmula se ha de usar 3.3 en vez
de 5.0 */
	voltaje = lectura * 5.0 / 1024.0;
	Serial.print(voltaje);
	Serial.println(" voltios");
//Convierto este voltaje en temperatura mediante la fórmula
	temperaturaC = (voltaje - 0.5) * 100 ;
	Serial.print(temperaturaC);
	Serial.println(" grados C");
	delay(1000);  //Para la estabilidad de las lecturas
}
```

A partir de aquí, se pueden realizar multitud de proyectos interesantes. Por ejemplo, podríamos tener un circuito formado por este sensor y tres LEDs, uno de color rojo, otro azul y otro verde. Cuando se detectara una temperatura más alta que la de un umbral superior predefinido, se podría iluminar el LED rojo, cuando la temperatura fuera más baja de un umbral inferior predefinido, se podría iluminar el LED azul y cuando la temperatura estuviera entre esos dos umbrales, se podría iluminar el LED verde. Se deja como ejercicio.

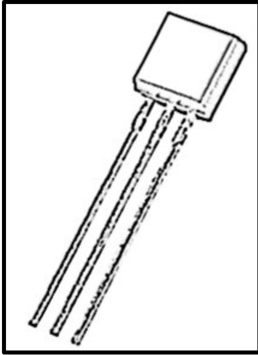
Ejemplo 7.16: Un código algo más elaborado que el del ejemplo anterior es el siguiente, en el que se asume un circuito con la presencia de, además del sensor TMP36, una pantalla LCD alfanumérica compatible con la librería oficial LiquidCrystal de Arduino y dos pulsadores (conectados a las entradas digitales nº 2 y nº 3 de la placa Arduino). La pantalla irá mostrando la temperatura actual continuamente excepto cuando se pulse un botón, momento en el que mostrará durante unos instantes la temperatura mínima y máxima registrada desde el arranque del sketch. Si se desea eliminar esos registros y obtener una nueva temperatura mínima y máxima a partir de la actual, se deberá apretar el otro botón.

```
#include <LiquidCrystal.h>
LiquidCrystal lcd(7, 8, 9, 10, 11, 12);
float Tmin = 0;
float Tmax = 0;
void setup() {
```



```
 //Podríamos haber hecho Tactual global, como Tmin o Tmax
  float Tactual=0;
  lcd.begin(16,2); lcd.clear();
  //Establezco un mínimo y máximo inicial a partir de la T actual
  Tactual=obtenerTemp();
  Tmin=Tactual; Tmax=Tactual;
  pinMode (2,INPUT); //Botón para redefinir la T máxima y mínima
  pinMode (3, INPUT); //Botón para mostrar la T máxima y mínima
}
void loop() {
  float Tactual=0;
  Tactual=obtenerTemp();
  //Si la temperatura actual supera los límites, los redefino
  if (Tactual<Tmin) { Tmin = Tactual; }
  if (Tactual>Tmax) { Tmax = Tactual; }
  mostrarTempActual(Tactual);  delay(100);
  //Si pulso el botón de "reset"
  if (digitalRead(2) == HIGH) { Tmin = Tactual; Tmax = Tactual; }
  //Si pulso el botón de mostrar máximo y mínimo
  if (digitalRead(3) == HIGH) {
    mostrarTempMin(); delay(3000);  mostrarTempMax(); delay(3000);
  }
}
/*Función que obtiene directamente la temperatura a partir
del voltaje leído en la entrada del sensor TMP36 */
float obtenerTemp(){ return (((analogRead(5)*5.0)/1024)-0.5)*100; }
/*Función que muestra la temperatura actual. Como Tactual es una
variable local de loop(), la tengo que pasar como parámetro. En
cambio, con Tmax y Tmin, al ser globales, eso no es necesario */
void mostrarTempActual(float Tactual){
   lcd.clear(); lcd.setCursor(0,0); lcd.print("Temperatura actual:");
   lcd.setCursor(0,1);  lcd.print(Tactual);  lcd.print(" C/");
}
//Función que muestra la temperatura mínima
void mostrarTempMin(){
   lcd.clear(); lcd.setCursor(0,0); lcd.print("Temperatura mínima:");
   lcd.setCursor(0,1);  lcd.print(Tmin);  lcd.print(" C/");
}
//Función que muestra la temperatura máxima
void mostrarTempMax(){
   lcd.clear(); lcd.setCursor(0,0); lcd.print("Temperatura máxima:");
   lcd.setCursor(0,1);  lcd.print(Tmax);  lcd.print(" C/");
}
```

El chip digital DS18B20 y el protocolo 1-Wire



El fabricante Maxim (anteriormente conocido como Dallas Semiconductor) produce una familia de componentes electrónicos que pueden ser controlados mediante un protocolo de comunicación propietario llamado “1-Wire”, el cual permite conectar a nuestra placa Arduino multitud de estos componentes mediante un solo cable de datos (de ahí su nombre). Otra característica destacable es que los componentes interconectados mediante este protocolo pueden estar situados a grandes distancias (de hasta incluso 30 metros).

El termómetro digital DS18B20 es un chip que utiliza el protocolo 1-Wire. Es muy popular debido a su bajo precio y facilidad de uso. Es capaz de medir temperaturas en un rango de -10 ^{o}C hasta 85 ^{o}C con una precisión de ±0,5 °C. En Sparkfun se puede encontrar con el código de producto nº 245, en Adafruit con el código nº 374 y en Freetronics podemos adquirirlo en forma de plaquita breakout, llamada “Temperature sensor module”, entre otros.

Si observamos de frente la cara lisa de su encapsulado, podemos observar que tiene tres patillas: la de más a la izquierda se corresponde con la conexión a tierra, la patilla central es la salida digital de la señal de datos (a conectar a una entrada digital de la placa Arduino) y la patilla derecha sirve para recibir la alimentación, la cual puede ser perfectamente los 5 V ofrecidos por la placa Arduino. Por tanto, en principio necesitaríamos tres cables para utilizar este componente. Además, es muy importante, para que el sensor funcione, conectar una resistencia de 4,7 KΩ entre la patilla de señal de datos y la patilla de alimentación.

No obstante, este chip (como el resto de componentes 1-Wire, de hecho) tiene la característica de poder conectarse de otra manera, utilizando tan solo dos cables. Esto es muy conveniente en sensores alejados cierta distancia de nuestra placa Arduino y/o situados en un entorno exterior. Es lo que se llama modo “parásito”, porque la alimentación que necesita la “parasita” de la señal de datos. Concretamente, las conexiones son: la patilla de la izquierda a tierra (como antes), la patilla central a una entrada digital de la placa Arduino (como antes) y la patilla derecha hay que unirla directamente a la patilla izquierda (para conectarla a tierra). Además, es muy importante conectar una resistencia de 4,7 KΩ entre el pin de entrada digital de la placa Arduino donde está conectada la patilla central y la alimentación de 5 V.

Otra característica muy interesante que ofrecen los componentes 1-Wire es la posibilidad de conectarse entre sí para formar una red (en nuestro caso, de sensores de temperatura) utilizando tan solo dos cables (es decir, aprovechando el modo “parásito”), independientemente del número de componentes conectados. El cableado en este caso es así: deberíamos conectar un primer sensor en modo parásito tal como lo hemos descrito en el párrafo anterior (incluyendo la resistencia de 4,7 KΩ), y los siguientes sensores han de tener todos su patilla izquierda conectada a la patilla izquierda del primer sensor (para mantener una misma tierra común), su patilla central conectada a la patilla central del primer sensor (para compartir el mismo –y único– cable de datos) y su patilla derecha conectada también a la patilla izquierda del primer sensor.

La conexión de múltiples componentes 1-Wire a un mismo cable es posible gracias a que cada uno de ellos tiene una dirección interna única formada por un código de 64 bits, dividido en un 8 bits para identificar el modelo de componente (el DS18B20 tiene el identificador 0x28), 48 bits para identificar ese componente individualmente en particular y 8 bits más (llamados código CRC) que sirven para comprobar que no haya errores en la identificación de ese componente por parte de los demás componentes del circuito (como por ejemplo, nuestra placa Arduino).

Para programar el DS18B20 mediante el lenguaje Arduino, podemos descargar de http://www.arduino.cc/playground/Learning/OneWire e instalar la librería “One-Wire”. Gracias a ella, la placa Arduino puede establecer comunicación con cualquier dispositivo diseñado para trabajar bajo el protocolo 1-Wire. No obstante, precisamente porque esta librería es genérica para cualquier dispositivo compatible con 1-Wire (y por tanto, muy flexible y versátil), es relativamente complicada de utilizar, ya que hay que conocer relativamente bien algunos detalles internos del protocolo 1-Wire. Si lo que deseamos es utilizar en concreto uno o más sensores de tipo DS18B20, tenemos la suerte de disponer de una librería específica, mucho más sencilla. Esta librería, llamada “Dallas Temperature Control Library ” y disponible en http://milesburton.com/Dallas_Temperature_Control_Library , necesita que se haya instalado previamente la librería “One-Wire” para funcionar, por lo que necesitaremos incluir las dos en nuestros sketches. Gracias a ella podremos leer el código identificador de cada sensor, configurar la resolución de las medidas (de 9 a 12 bits) y obtener el valor de la temperatura de cada uno de los sensores de una forma muy simple.

Ejemplo 7.17: A continuación, se muestra un código de ejemplo, donde se ha supuesto que la patilla de datos de un sensor está conectado al pin digital de entrada nº 2 de la placa Arduino:

```
#include <OneWire.h>
#include <DallasTemperature.h>
/*Creo un objeto de tipo “OneWire”. Esta instrucción pertenece a la
librería One-Wire, por lo que es genérica para cualquier dispositivo
compatible con ese protocolo, no solamente el DS18B20. El valor del
parámetro indica el número de pin digital de la placa Arduino donde
está conectada la patilla de datos del sensor (o sensores). */
OneWire dispositivoOneWire(2);
/*Pasamos la referencia del objeto recién
creado a la librería DallasTemperature */
DallasTemperature sensores(&dispositivoOneWire);
setup(){
	Serial.begin(9600);
/*Se inicializa la librería. Opcionalmente, se puede especificar un
parámetro para ajustar la precisión de las medidas: por defecto es de
9 bits, pero puede aumentar hasta 12. La contrapartida es una mayor
lentitud en las lecturas*/
	sensores.begin();
}
loop(){
/*Solicito las temperaturas de todos los
posibles sensores presentes en el canal de datos */
	sensores.requestTemperatures();
	Serial.print("Temperatura para el dispositivo 1 es: ");
/*El parámetro de getTempCByIndex indica el número de sensor del cual
se quiere obtener la temperatura en grados Celsius: el “0” se
corresponde al primer sensor presente en el cable de señal de datos,
el “1” sería el siguiente, y así*/
	Serial.print(sensores.getTempCByIndex(0));
}
```

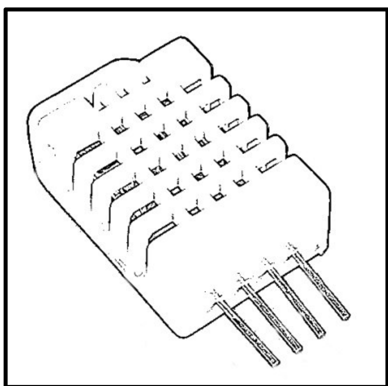
La plaquita breakout TMP421

Modern Device distribuye una plaquita que incluye el chip analógico TMP421. Debemos conectarla a los pines analógicos nº 2, 3, 4 y 5 de la placa Arduino. El primer servirá para alimentar la placa, el segundo para conectarla a tierra, y los dos últimos sirven para establecer el canal I^2C a través del cual se realizará la comunicación entre ambos dispositivos. Puede medir temperaturas desde -40 ^oC hasta 125 ^oC con un error de ± 1 ^oC. Es realmente fácil de usar mediante la librería disponible en https://github.com/moderndevice/LibTempTMP421.

SENSORES DE HUMEDAD

El sensor DHT22/RHT03

En este apartado hablaremos concretamente del sensor digital de temperatura y humedad RHT03 de Maxdetect (http://www.humiditycn.com) , el cual se distribuye (además de muchos otros) en Sparkfun con nº de producto 10167 y en Adafruit (aquí con el nombre de DHT22 y con nº de producto nº 385). También se le puede encontrar con el nombre de AM2302. Este sensor es muy básico y lento (solo se pueden obtener datos como mínimo cada 2 segundos), pero es de bajo coste y muy manejable para obtener datos básicos en proyectos caseros.



Sus características técnicas más destacables son: se puede alimentar con un voltaje de entre 3 V y 5 V y 2,5 mA como máximo, puede medir un rango de temperaturas entre -40 y 125 °C con una precisión de ±0,5 °C y un rango de humedad entre 0 y 100% con una precisión del 2-5%. Está formado básicamente por un sensor de humedad capacitivo y un termistor.

Este chip tiene cuatro pines; mirándolo de frente son: el de alimentación (nº 1, el de más a la izquierda), el de salida digital de datos (nº 2), uno no conectado a nada y que se puede ignorar (nº 3) y el de tierra (nº 4, el de más a la derecha). Así pues, para que nuestra placa Arduino pueda leer los datos que emite este chip, deberemos conectar su pin nº 1 al pin-hembra de 5 V de la placa (por ejemplo), su pin nº 4 a un pin "GND" y su pin nº 2 a una pin-hembra de entrada digital. Además, es recomendable conectar una resistencia ("pull-up") de 4,7 KΩ entre el pin nº 1 y nº 2.

Desgraciadamente, el protocolo que utiliza este sensor para transmitir los datos digitales no es estándar, así que en principio deberíamos de aprender su funcionamiento interno para poder interpretar correctamente la información que nos esté llegando en un momento determinado. Por suerte, tenemos a nuestra disposición gran cantidad de librerías Arduino, compatibles con este sensor. Estas librerías, a pesar de ser diferentes entre sí, son en gran medida equivalentes, ya que todas ellas lo que hacen básicamente es permitir centrarnos en la obtención sencilla de los datos de temperatura y humedad sin tener que conocer los detalles específicos del protocolo particular utilizado por el chip. Por lo tanto, la elección de una librería u otra no es determinante.

Ejemplo 7.18: La gente de Adafruit, por ejemplo, ha programado una librería Arduino compatible con este sensor (https://github.com/adafruit/DHT-sensor-library) Una vez instalada como cualquier otra librería, podemos probarla ejecutando el siguiente sketch de ejemplo, el cual nos muestra por el "Serial monitor" la temperatura y humedad ambiente. Para ver cambios en los datos obtenidos, se puede probar de expulsar vaho sobre el sensor si se quiere aumentar la humedad leída:

```
#include <DHT.h>
/*El primer parámetro es el número del pin-hembra de la placa Arduino
donde está conectado el pin de datos del sensor. El segundo dato
especifica el modelo de sensor que se está utilizando (ya que la
librería puede servir para varios. El valor correspondiente para el
DHT22 es "22". Para consultar los otros valores posibles, se puede
mirar el archivo "DHT.h" que viene incluido dentro de la librería.*/
DHT dht(2, 22);
void setup() {
       Serial.begin(9600);
       dht.begin();
}
void loop() {
      float h,t;
      //Las lecturas del sensor pueden tardar hasta 2 segundos
       h = dht.readHumidity();
       t = dht.readTemperature();
/*Compruebo que los datos recibidos son válidos. Concretamente, la
función isnan() mira si el dato no es un número ("IS Not A Number"),
en cuyo caso algo ha ido mal*/
      if (isnan(t) || isnan(h)) {
             Serial.println("Algo falló ");
       } else {
             Serial.print("Humedad: ");
             Serial.print(h);
             Serial.print(" %\t");
             Serial.print("Temperatura: ");
             Serial.print(t);
             Serial.println(" *C");
      }
}
```

Ejemplo 7.19: Si lo que queremos es (en vez de mostrar los datos obtenidos por el canal serie) guardarlos en una tarjeta SD para poderlos estudiar con calma

posteriormente, deberíamos modificar el sketch anterior para que se parezca al siguiente. En él hemos añadido comprobaciones extra para tener en cuenta los posibles errores que pueden ocurrir en la lectura en la tarjeta:

```
#include <SD.h>
#include <DHT.h>
DHT dht(2,22);
const int intervalo = 10*1000; //Intervalo entre lecturas, en ms
long tiempoUltimaLectura = 0;  //Tiempo de la última lectura, en ms
long tiempoActual = 0;         //Tiempo actual devuelto por millis()
float h,t;                     //Humedad y temperatura obtenidas
void setup() {
  /*Utilizo una función propia para inicializar la tarjeta SD. Si la
  tarjeta se inicializa correctamente, inicializo la librería DHT */
  if (inicioSD() == true) {
     dht.begin();
  }
}
void loop(){
  /*Uso el truco de repetir código cada determinado tiempo
  (especificado por "intervalo") sin utilizar delay() */
  tiempoActual = millis();
  if (tiempoActual > tiempoUltimaLectura + intervalo) {
    //Obtengo los datos de temperatura y humedad
    h = dht.readHumidity();
    t = dht.readTemperature();
    //Si las lecturas NO son erróneas (atención al "!")
    if (!isnan(t) || !isnan(h)) {
       //Abro el fichero y escribo a partir de la última línea
        File mifichero = SD.open("datalog.csv", FILE_WRITE);
        if (mifichero==true) {
            mifichero.print(h);
            mifichero.print("\t");  //Tabulador
            mifichero.println(t);
            mifichero.close();
            tiempoUltimaLectura = millis();
        }
    }
}
boolean inicioSD() {
  boolean resultado = false;
  //Aunque no se utilice, el pin 10 se ha de configurar como salida
  pinMode(10, OUTPUT);
  //Suponemos que la tarjeta está conectada al canal SS vía pin 4
```

```
  if (SD.begin(4)==false) {
    //Si no lo está, el sketch no hará nada
    resultado = false;
  }
  /*Si sí, abro el fichero "datalog.csv" para escribir una línea
  (lo que sería el título) a continuación de las posibles líneas
  anteriores que existan de otras ejecuciones previas.*/
  else {
    File mifichero = SD.open("datalog.csv", FILE_WRITE);
    if (mifichero == true) {
      mifichero.println();
      mifichero.println("rH (%) \t temp. (*C)");
      mifichero.close();
      resultado = true;
    }
  }
  return resultado;
}
```

Lo interesante de los dos sketches anteriores es que ambos envían (uno al canal serie y el otro a la tarjeta SD) los datos de temperatura y humedad de forma tabulada, con incluso un título para las columnas. Este hecho lo podemos aprovechar para importar fácilmente esta información en programas de procesamiento de datos, (como pueden ser las hojas de cálculo), para realizar así cálculos estadísticos o representación de gráficas, entre otras posibilidades.

En el caso del fichero guardado en la tarjeta SD, para realizar esta importación lo único que deberemos hacer es colocar dicha tarjeta en un lector conectado a nuestro computador y abrir ese fichero con nuestro programa preferido, como puede ser el software LibreOffice Calc (http://www.libreoffice.org, aplicación de hoja de cálculo libre, multiplataforma y gratuita) o, si tan solo se requiere el dibujo de gráficas y nada más, el software LiveGraph (http://www.live-graph.org, graficador también libre, multiplataforma y gratuito).

En el caso de obtener los datos a través del “Serial monitor”, lo que deberemos hacer es lo siguiente: pulsar dentro de él la combinación de teclas CTRL+A para seleccionar todos los datos y pulsar CTRL+C para copiarlos en memoria. A partir de aquí, podemos abrir un nuevo fichero de texto plano y pulsar CTRL+V para pegar los datos, o bien abrir una nueva hoja de cálculo usando por ejemplo LibreOffice Calc. En este último caso, una vez ya dentro de la hoja de cálculo, al pulsar también CTRL+V para pegar los datos veríamos como en ese momento aparece un cuadro emergente

de configuración de la importación de texto. En ese cuadro deberíamos seleccionar la opción “separado por tabulador” (o similar) para que los datos fueran correctamente incorporados.

Si utilizáramos otro programa diferente del “Serial monitor” para ver los datos recibidos por el canal serie, podríamos transferir estos a una hoja de cálculo de una forma más sencilla, ya que la gran mayoría de ellos tienen la opción de guardar a tiempo real los datos recibidos en un fichero de texto listo para ser importado.

Otra manera de procesar los datos leídos por el canal serie que permita realizar un análisis posterior sobre ellos es utilizar un programa gratuito llamado MegunoLink (http://www.blueleafsoftware.com). Este programa es capaz (entre otras funcionalidades) de dibujar por pantalla y en tiempo real una gráfica correspondiente a los datos recibidos por el canal serie. Para que funcione, deberemos utilizar dentro de nuestros sketches una librería que nos proporcionan los mismos desarrolladores de MegunoLink, la llamada “Arduino graphing library”. Junto con la librería se ofrecen varios códigos de ejemplo donde se puede ver su comportamiento; allí se puede observar cómo podemos configurar las nombres de los ejes X e Y de la gráfica y sus diferentes leyendas. Desgraciadamente, MegunoLink solo funciona en sistemas Windows. Otra graficador en tiempo real (que también funciona solo para Windows) es SerialChart (http://code.google.com/p/serialchart).

Volviendo a los sensores de humedad, otra librería compatible con el sensor DHT22/RHT03 es la que podemos encontrar en https://github.com/ringerc/Arduino-DHT22. Otra más es la disponible en http://arduino.cc/playground/Main/DHTLib. Por otro lado, Freetronics distribuye una plaquita breakout con el mismo sensor DHT22/RHT03 (bajo el nombre de “Humidity and Temperature Sensor Module”), que ofrece tres conectores: tierra, señal de datos y alimentación (listados de izquierda a derecha) y se puede programar mediante una librería de la propia Freetronics, disponible en https://github.com/freetronics/DHT-sensor-library, muy parecida a la de Adafruit.

Los sensores SHT15 y SHT21

Otro sensor de humedad (y temperatura) es el SHT15, distribuido por Sparkfun en forma de plaquita breakout con código de producto 8257. Esta plaquita consta de cuatro pines: alimentación (5 V), tierra, conector “SCK” y conector “DATA”. Estos dos últimos han de conectarse a dos entradas digitales cualesquiera de nuestra placa Arduino. Aunque el protocolo de comunicación que utiliza este chip requiere el uso de dos cables al igual que pasa con I^2C, no hay que confundirlos porque son

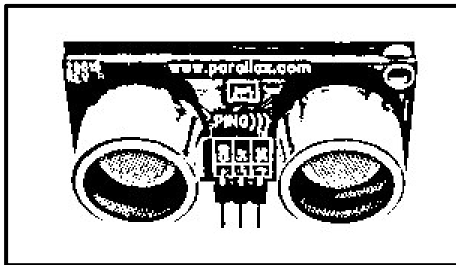
sistemas de comunicación diferentes. Como datos técnicos a destacar, podemos decir que alcanza una precisión de hasta $\pm$0,3 ^{o}C en medidas de temperatura y $\pm$2 % en medidas de humedad y tiene un tiempo de respuesta menor de 4 segundos.

La manera más sencilla de programar este sensor es utilizando la librería disponible en https://github.com/practicalarduino/SHT1x (también válida para otros sensores de la misma familia fabricada por Sensirion, como el SHT10, el STH11, el SHT71 o el SHT75).

Por otro lado, un chip del mismo fabricante que sí es capaz de utilizar la comunicación I^2C es el sensor SHT21. LoveElectronics distribuye una plaquita breakout que ofrece de forma muy cómoda los cuatro conectores necesarios: alimentación (3,3 V, atención), tierra, pin SDA y pin SCL. Se puede programar mediante la librería (también válida para el SHT25) disponible en https://github.com/misenso/SHT2x-Arduino-Library. Otra plaquita muy parecida la distribuye Modern Device, junto con la librería "LibHumidity", descargable de la web del producto.

SENSORES DE DISTANCIA

El sensor Ping)))



El sensor digital de distancia Ping)))™ de Parallax es capaz de medir distancias entre aproximadamente 3 cm y 3 m. Esto lo consigue enviando un ultrasonido (es decir, un sonido de una frecuencia demasiado elevada para poder ser escuchado por el oído humano) a través de un transductor (uno de los cilindros que se aprecian en la figura lateral) y espera a que este ultrasonido rebote sobre un objeto y vuelva, retorno que es detectado por el otro transductor. El sensor devuelve el tiempo transcurrido entre el envío y la posterior recepción del ultrasonido. Como la velocidad de propagación de cualquier (ultra)sonido en un medio como el aire es de valor conocido (consideraremos que es de 340 m/s –o lo que es lo mismo, 0,034 cm/µs–, aunque esta sea solo una aproximación) este tiempo transcurrido lo podremos utilizar para determinar la distancia entre el sensor y el objeto que ha provocado su rebote.

La plaquita en la que viene muestra tres pines marcados. Teniendo visibles enfrente los transductores son, de izquierda a derecha: el pin de tierra, el de

alimentación (5 V) y el pin para comunicarse con un pin-hembra digital de nuestra placa Arduino (lo llamaremos a partir de ahora “pin de señal”).

Este sensor solo mide distancias cuando se le solicita. Para ello, nuestra placa Arduino envía a través de su pin digital (conectado al pin de señal de la plaquita) un pulso HIGH de una duración exacta de 5 microsegundos. Esta es la señal que activa el envío del ultrasonido. Tras un breve lapso de tiempo, el sensor recibirá el rebote del ultrasonido y como consecuencia, nuestra placa Arduino recibirá ese dato por el mismo pin digital utilizado anteriormente. En ese momento, la placa Arduino podrá calcular el tiempo transcurrido entre el envío y la recepción de ese ultrasonido. El funcionamiento descrito obliga a que se deba alternar el modo del pin digital de la placa Arduino conectado al sensor, de forma que sea de tipo INPUT o OUTPUT según convenga.

Para calcular la distancia, debemos recordar la fórmula **v = d/t** (que no es más que la definición de velocidad: distancia recorrida en un determinado tiempo). Si de la fórmula anterior despejamos la “d” (la distancia recorrida) obtenemos finalmente d = v·t, donde “v” la conocemos y la consideramos constante (0,034 cm/µs) y “t” es el tiempo medido por el sensor. Hay que tener en cuenta, no obstante, que el dato obtenido del sensor es el tiempo total que tarda el ultrasonido en “ir y volver”, así que normalmente querremos dividir previamente este valor entre dos para asignárselo a “t”.

Ejemplo 7.20: El siguiente sketch realiza todos estos cálculos y muestra el resultado a través del canal serie. Hemos supuesto que el pin de señal de la plaquita está conectado al pin-hembra digital nº 8 de Arduino. Se puede probar en tiempo real poniendo un obstáculo enfrente del sensor y moviéndolo adelante y atrás.

```
int distancia;
unsigned long tiempo=0;
void setup(){
	Serial.begin(9600);
}
void loop(){
	//Obtengo la lectura de la distancia medida por el sensor
	enviarYRecibir();
	/*Convertimos el tiempo a distancia, sabiendo
	la velocidad del ultrasonido (en cm/microsegundos) */
	distancia = int(0.034*tiempo);
	Serial.print("Distancia: ");
	Serial.print(distancia);
```

```
        Serial.println(" cm");
        delay(500);
}
void enviarYRecibir(){
/*Configuramos el pin de datos como
salida y estabilizamos el sensor para
poder enviarle el pulso de activación*/
        pinMode(8, OUTPUT);
        digitalWrite(8, LOW);
        delayMicroseconds(5);
/*Ahora es cuando enviamos el pulso
de 5 microsegundos para activar el sensor
y enviar el ultrasonido */
        digitalWrite(8, HIGH);
        delayMicroseconds(5);
        digitalWrite(8, LOW);
/*Mientras el ultrasonido viaja, cambiamos
el modo del pin de datos a entrada
para leer el rebote del pulso */
        pinMode(8, INPUT);
/*Medimos la longitud del pulso entrante. El truco aquí está en saber
que justo después de enviar el ultrasonido, el sensor comienza
siempre a emitir una señal HIGH, la cual será recibida por el pin de
la placa Arduino acabado de configurar como entrada (el n° 8 en este
caso). Pero en el momento que se reciba el rebote, el sensor
automáticamente cambiará esa señal HIGH a LOW. Esto permite utilizar
la función pulseIn() para medir el tiempo transcurrido entre un
instante y otro: tal como está escrita en el sketch, esta función
cuenta el tiempo que pasa desde que el pin n° 8 empieza recibiendo
una señal de valor HIGH (en el momento de enviar el pulso) hasta que
lo deja de recibir (en el momento de recibir el rebote). La duración
de esa señal HIGH recibida se corresponde con el tiempo que queremos
medir*/
        tiempo=pulseIn(8, HIGH);
/*Divido la longitud del pulso a la mitad, porque solo quiero
utilizar el tiempo tardado en la ida del ultrasonido, no el
de ida y vuelta*/
        tiempo=tiempo/2;
}
```

A partir de la detección de la presencia de un objeto, las reacciones del circuito pueden ser muy diversas: pueden consistir en la activación de alarmas acústicas o lumínicas, o en la apertura o cierre de diferentes compuertas (a través de servomotores) o en cualquier otra cosa que la imaginación sugiera. Por ejemplo,

como posible ejercicio proponemos la construcción de un radar casero: solo tendríamos que situar el sensor sobre un servomotor que continuamente estuviera moviéndose entre 0 y 180 grados (y viceversa). En el momento que se detectara algún objeto a una distancia menor de la establecida como umbral, se podría iluminar un LED o activar un zumbador, por ejemplo.

El sensor SRF05

Otro sensor digital que utiliza el método de contar el tiempo transcurrido entre la emisión de un pulso ultrasónico y su posterior recepción para medir distancias es el SRF05 de Devantech. Es capaz de medir distancias entre 3 cm y 3 m a un ritmo de hasta 20 veces por segundo.

Este sensor se puede conectar de dos maneras diferentes a nuestra placa Arduino: o bien utilizando cuatro cables ("modo 1" compatible con su predecesor, el sensor SRF04), o bien usando tres ("modo 2"). Si observamos el dorso del sensor y mantenemos la serigrafía "SRF05" visible a nuestra izquierda, en el modo 1 los cinco conectores de la zona inferior se corresponden, de izquierda a derecha, con: alimentación (5 V), entrada del rebote ultrasónico (pin "echo"), salida del pulso ultrasónico (pin "trigger"), pin que no se ha de conectar a nada y tierra. El pin "echo" se ha de conectar a un pin de entrada digital de nuestra placa Arduino y el pin "trigger" a un pin de salida digital. Este pin "trigger" es el responsable de generar una pulso con una duración exacta de 10 microsegundos, el cual marca el inicio del envío de la señal ultrasónica, y el pin "echo" utiliza el mismo truco que el sensor Ping))) para contar el tiempo transcurrido entre envío y recepción del ultrasonido: el mantener una señal HIGH mientras no se reciba el rebote.

Si utilizamos el modo 2, estos mismos conectores se corresponden (de izquierda a derecha también) con: alimentación (5 V), pin que no se ha de conectar a nada, salida y entrada todo en uno de la señal ultrasónica, tierra y tierra otra vez. En este modo, el sensor utiliza un solo pin para enviar el pulso y recibir el rebote (tal como ocurre de hecho con el sensor Ping)))). Esto permite utilizar un cableado más simple, aunque la programación se complica, porque el mismo pin ha de alternar entre ser entrada y salida dependiendo de las circunstancias.

<u>Ejemplo 7.21</u>: El código siguiente, que utiliza el modo 1, gradúa el brillo de un LED enviándole una señal PWM que variará según lo cercano que esté un obstáculo al sensor: la idea es que cuanto más cerca esté, más brille el LED. Concretamente, a cada centímetro (por debajo de los 255 cm) más próximo se incrementa el brillo del LED en un punto más de la señal PWM. Si se deseara utilizar el rango máximo de

distancias que permite el sensor, el brillo del LED se debería calcular de otra manera, (como por ejemplo en función del tanto por ciento de la distancia medida respecto a la distancia máxima posible), pero tal como se presenta aquí, el código es más sencillo.

Otro detalle del código digno de comentar es que, para evitar obtener ruido (es decir, valores individuales muy fluctuantes) y por tanto conseguir lecturas más suavizadas, se ha utilizado un sistema ya visto anteriormente en otras ocasiones: el cálculo de la media de las últimas mediciones realizadas.

También se puede observar que para obtener la distancia, el tiempo transcurrido entre la señal de “trigger” y de “echo” (que es, en definitiva, lo que mide el sensor, calculado en microsegundos) se ha multiplicado por 0,017. ¿Por qué? Porque tal como se explicó cuando se vio el sensor Ping))), se ha hecho servir la fórmula básica d = v·t, donde v = 0,034 (la velocidad de un ultrasonido en el aire en unidades de cm/μs ,o lo que es lo mismo, 340 m/s) y “t” se ha de dividir entre 2 para contar solo el trayecto de “ida” y no de “ida y vuelta”.

```
const int nLecturas=10; //Número de lecturas para hacer media
int lecturas[nLecturas]; //Array que guarda las últimas lecturas
int indice = 0;         //Posición actual dentro del array
int total = 0;          //Suma de las lecturas guardadas en el array
int media = 0;          //Media de las lecturas guardadas en el array
unsigned long tiempo = 0;
unsigned long distancia = 0;
void setup() {
  pinMode(9, OUTPUT); //Donde está conectado el LED (señal PWM)
  pinMode(2, INPUT);  //Donde está conectado el pin “echo” del sensor
  pinMode(3, OUTPUT); //Donde está conectado su pin “trigger”
  //Creo un array inicialmente vacío
  for (int i = 0; i < nLecturas; i++) {lecturas[nLecturas] = 0;}
  Serial.begin(9600);
}
void loop() {
  //Estabilizamos el sensor antes de enviar el pulso de activación
  digitalWrite(3, LOW);
  delayMicroseconds(5);
  digitalWrite(3, HIGH); //Envío el pulso de activación
  delayMicroseconds(10); //La duración de este pulso ha de ser 10 μs
  //Paro la activación y comienza el envío de la señal ultrasónica
  digitalWrite(3, LOW);
  /*Inmediatamente después de empezar el envío del ultrasonido, por
```



```
  el pin “echo” se empieza a recibir una señal HIGH. La función
  pulseIn() pausa entonces el sketch para contar el tiempo
  transcurrido hasta recibir el rebote, momento en el cual por el pin
  “echo” pasa a detectarse una señal LOW y pulseIn() devuelve su
  resultado.*/
  tiempo = pulseIn(2, HIGH);
  /*Calculo la distancia (distancia en cm = velocidad en cm/µs
  multiplicado por el tiempo en µs). Ya se ha dividido entre dos para
  contar solo el tiempo de ida*/
  distancia = 0.017*tiempo;
  /*Pero no quiero obtener un valor de distancia individual, porque
  puede ser muy fluctuante: quiero que el valor tenido en cuenta sea
  una media de los últimos diez valores individuales medidos. Para
  ello, primero elimino de la suma total el valor ubicado en el
  índice actual del array, que corresponde a la medida tomada hace
  diez iteraciones */
  total= total - lecturas[indice];
  /*Ahora añado la nueva distancia medida precisamente en ese
  elemento del array, sobrescribiéndolo. De esta manera, se hace una
  reescritura cíclica de cada elemento cada diez lecturas*/
  lecturas[indice] = distancia;
  //Y finalmente, añado este nuevo valor a la suma total otra vez
  total= total + lecturas[indice];
  indice = indice + 1; //Voy al siguiente elemento del array
  //Al final del array (10 elementos) vuelvo a empezar
  if (indice >= nLecturas)  {  índice = 0;  }
  media = total / nLecturas;  //Calculo la media
  /*Si la distancia es menor que 255 cm cambio el brillo del LED, de
  forma que a menor distancia más brillante sea */
  if (media < 255) {
     analogWrite(9, 255 - media);
  }
  delay(100); //El tiempo mínimo entre lecturas ha de ser de 20µs
}
```

Si quisiéramos utilizar el modo 2, simplemente tendríamos que haber sustituido las primeras líneas de la función “loop()” del sketch anterior (hasta la línea de *pulseIn(),* esta incluida) por las siguientes (donde suponemos que el pin “trigger-echo” del sensor está conectado al pin nº 2 de la placa Arduino):

```
/*Mandamos un pulso bajo de 5 microsegundos para asegurar que siempre
se inicie en LOW, y seguidamente mandamos una señal HIGH que sirve
para iniciar las mediciones*/
pinMode(2, OUTPUT);
```

```
digitalWrite(2, LOW);
delayMicroseconds(5);
digitalWrite(2, HIGH);
delayMicroseconds(10);
digitalWrite(2, LOW);
/*Como utilizamos el mismo pin para recibir el eco, lo cambiamos de
salida a entrada*/
pinMode(2, INPUT);
tiempo = pulseIn(2, HIGH);
```

El sensor HC-SR04

Este sensor es muy parecido a los anteriores. Dispone de cuatro pines: "VCC" (se ha de conectar a una fuente de 5 V), "Trig" (responsable de enviar el pulso ultrasónico; por tanto, se deberá conectar a un pin de salida digital de la placa Arduino), "Echo" (responsable de recibir el eco de ese pulso; luego se deberá conectar a un pin de entrada digital de la placa Arduino) y "GND" (a tierra). Se puede adquirir en IteadStudio o Elecfreaks por menos de diez euros.

Al igual que los anteriores sensores, tiene un rango de distancias sensible entre 3 cm y 3 m con una precisión de 3 mm, y su funcionamiento también es muy parecido: tras emitir por el pin "trigger" una señal de 10 µs para iniciar el envío de la señal ultrasónica, espera a detectar el eco mediante la detección del fin de la señal HIGH recibida por el pin "echo".

De hecho, el código de ejemplo mostrado para el modo 2 del sensor SRF05 puede ser utilizado sin ningún tipo de cambio en este sensor. De todas maneras, si se desea, se puede utilizar la librería "New-Ping", descargable desde http://code.google.com/p/arduino-new-ping, la cual ofrece una manera sencilla y común de gestionar diferentes modelos de sensores de distancia ultrasónicos, como por ejemplo el propio HC-SR04 (pero también el sensor Ping))) y el SRF05, entre otros).

El sensor LV-EZ0

Otro sensor de distancia que utiliza ultrasonidos es el sensor LV-EZ0 de Maxbotix. No obstante, a diferencia de los anteriores, el LV-EZO es un sensor analógico. Por ello, para usarlo con nuestra placa Arduino deberemos conectar (además del pin "+5 V" a la alimentación de 5V proporcionada por la placa Arduino y del pin "GND" a la tierra común) el pin etiquetado como "AN" a una entrada analógica de nuestra placa Arduino.

El rango de distancias que puede medir este sensor depende mucho del tamaño del obstáculo: si este es del tamaño de un dedo, el rango es aproximadamente de 2,5 metros; si este es del tamaño de una hoja de papel, el rango puede aumentar hasta 6 metros. En todo caso, no es capaz de detectar distancias más pequeñas de 30 cm. La buena noticia está en que el comportamiento de este sensor es lineal: si un obstáculo está por ejemplo a 2 metros, la lectura de tensión recibida por el pin de entrada analógica será la mitad que si está a 4 metros. Esto permite que las lecturas sean muy fáciles de realizar.

Ejemplo 7.22: Podemos utilizar un sketch tan simple como el siguiente para observar las diferentes distancias detectadas de un obstáculo a lo largo del tiempo.

```
int sensorPin = 0; //El sensor está conectado al pin analógico n° 0
void setup(){
  Serial.begin(9600);
}
void loop(){
  Serial.println(analogRead(sensorPin));
  /*El delay() ralentiza la toma de lecturas para hacerlas más
fáciles de leer. No obstante, si queremos detectar la presencia de
obstáculos que rápidamente atraviesan de un lado al otro el espacio
sensible, este delay() se podría reducir o quitar */
  delay(100);
}
```

¿Cómo se podría modificar el código anterior para que, añadiendo un LED conectado a un pin de salida digital de nuestra placa Arduino, este se encendiera cuando se detectara la presencia de un obstáculo más cerca de una determinada distancia umbral? Se deja como ejercicio.

Existen varias versiones de este sensor con diferentes rangos de distancia detectables y distintas anchuras del espacio sensible dentro del cual se pueden reconocer los obstáculos. El sensor EZ1 tiene un espacio sensible más estrecho que el EZ0, el EZ2 lo tiene más estrecho que el EZ1, y así hasta el EZ4. Un espacio sensible estrecho es mejor si tan solo se desean detectar objetos directamente situados enfrente del sensor, y uno ancho es mejor si se quiere detectar cualquier objeto cercano. Maxbotix también ofrece una línea de sensores más precisos (los “XL”) que tienen hasta una precisión de 1 cm, más rango de distancia y mejor supresión de ruido (es decir, que sus lecturas son menos tambaleantes). Sparkfun distribuye el sensor EZ0 con código de producto 8502, Adafruit con el código 979. Si se quieren encontrar otros sensores fabricados por Maxbotix en las tiendas online de estos distribuidores, basta con escribir “Maxbotix” en el buscador.

Los sensores GP2Yxxx

Existe una familia de sensores analógicos de distancia fabricados por Sharp y cuyo nombre empieza por GP2Y que, a diferencia de los anteriormente descritos, utilizan ondas infrarrojas. Su funcionamiento genérico es el siguiente: el chip contiene en su parte frontal un emisor y un receptor infrarrojos. Si el haz infrarrojo (que está modulado) impacta con un obstáculo, se dispersará en varias direcciones, pero también en la dirección donde está situado precisamente el receptor. Utilizando el cálculo de triangulaciones, a partir de la medida del ángulo de incidencia del haz infrarrojo sobre el receptor, se puede deducir la distancia del obstáculo. La salida de estos sensores será una tensión proporcional a la distancia calculada.

Este método tiene algunos inconvenientes. Por ejemplo: la luz del sol (que contiene infrarrojos), tanto directa como reflejada en grandes cantidades puede perturbar la lectura. Para minimizar este problema, es una buena práctica no utilizar lecturas individuales sino tomar un promedio de un número de valores como la lectura válida. Otro inconveniente que hay que tener en cuenta es el de la distancia mínima de nuestro sensor: existe un ángulo máximo más allá del cual los sensores infrarrojos ya no recibe correctamente el reflejo del haz, por lo que los valores medidos a distancias bajo ese mínimo tienen poco significado.

Elegiremos para su estudio concretamente el sensor GP2Y0A02. Este sensor lo podemos encontrar por ejemplo en Pololu con código de producto 1137. Su rango de medida está entre 20 y 150 centímetros y como conector incorpora uno de tipo JST de 3 pines, por lo que para enchufarlo a una breadboard necesitaríamos un cable tal como el producto nº 117 de Littlebird Electronics o el nº 8733 de Sparkfun, el cual consta por un lado de un zócalo JST de 3 pines y por otro están los tres cables libres sin terminal. Cada pin del sensor se corresponde con la alimentación (5 V), tierra y la salida analógica (a conectar a un pin-hembra de entrada analógica de la placa Arduino).

Este sensor devuelve un voltaje que es inversamente proporcional a la distancia: cuanta más distancia haya con el objeto, menos voltaje genera el chip (aunque la relación no es lineal). Este hecho lo podemos aprovechar para escribir sketches compuestos por un conjunto de "ifs" del estilo "si el voltaje medido está en un rango determinado de valores que ocurra algo y si está en otro, que ocurra otra cosa" si lo único que nos interesa es comparar diferentes distancias sin necesidad de saber su valor exacto.

Si lo que queremos, en cambio, es conocer la distancia real del obstáculo, podemos consultar en el datasheet del sensor un gráfico preciso del voltaje proporcionado por este en función de la distancia medida, por lo que a partir de los

valores mostrados en esta gráfica podríamos interpretar los valores de voltaje recibidos y mostrarlos como distancia.

Ejemplo 7.23: El siguiente sketch, no obstante, utiliza una fórmula para obtener la distancia. Esta fórmula se obtiene precisamente de los datos de la gráfica de su datasheet, pero no es exacta, así que hay que tomarla con precaución. Posiblemente sea necesario calibrar los datos numéricos que intervienen (en nuestro caso, el 65 y el -1,1) para acabar de perfilar las mediciones en nuestro caso particular.

```
float sensor = 0.0;
float distancia = 0.0;
void setup(){
       Serial.begin(9600);
}
void loop(){
      sensor = analogRead(0);
/*Convierto el dato obtenido por el conversor analógico-digital en un valor de voltaje. Si el sensor se alimenta con 3,3V, hay que sustituir el 5.0 por un 3.3 */
      sensor = sensor * 5.0 / 1024.0;
//Calculo la distancia a partir del voltaje obtenido
      distancia = 65*pow(sensor, -1.10);
      Serial.print(distancia);
      delay(250);
}
```

Una utilidad muy curiosa de los sensores de distancia es construir theremines caseros. Un theremin es un instrumento musical que posee un par de antenas metálicas que controlan el volumen y la frecuencia de la nota emitida, según nos acerquemos o alejemos de ellas. En lugar de dos antenas metálicas, utilizaremos un sensor de distancia; concretamente el G2PY0A02 (aunque lógicamente también podríamos haberlo implementado con otros modelos, tales como los ultrasónicos). La idea es obtener un valor interpretable como distancia a partir del valor de tensión leído de este sensor, y a partir de allí generar el sonido correspondiente.

Ejemplo 7.24: Una implementación concreta de theremin es el sketch siguiente, el cual reproduce diferentes notas musicales estándares según sea la distancia detectada (cuanto más cerca esté el obstáculo –nuestra mano–, más aguda será la nota). La onda de sonido se emite a través de un zumbador conectado al pin digital de salida nº 8 de nuestra placa Arduino. Ese sonido ha de durar un tiempo lo suficientemente breve (hemos puesto 125 milisegundos pero se puede cambiar) para

poder hacer un seguimiento en tiempo real de la detección del movimiento del obstáculo.

```
int lecturaSensor, nota;
//Array que guarda las frecuencias a reproducir
float frecuencias[] = {
  329.63, // Mi
  349.23, // Fa
  369.99, // Fa#
  392.00, // Sol
  415.30, // Sol#
  440.00, // La
  466.17, // La#
  493.83, // Si
  523.25, // Do
  554.36, // Do#
  587.33, // Re
  622.25, // Re#
  659.26, // Mi
};
byte numFrecuencias=13; //Número de elementos del array
void setup(){
       Serial.begin(9600);
       pinMode(8,OUTPUT); //Donde está conectado el zumbador
}
void loop() {
/*Cuanto más lejos esté el obstáculo, “lecturaSensor” menos valdrá
De todas formas, hay que saber que esta relación no es lineal*/
      lecturaSensor=analogRead(0); //Donde está conectado el sensor
/*Asocio “lecturaSensor” a una de las frecuencias predefinidas*/
      nota=map(lecturaSensor,0,1023,1,numFrecuencias);
/*Hago que a medida que el obstáculo se acerca, la nota es más
aguda*/
      tone(8,frecuencias[nota]);
/*Tiempo de reproducción mínimo de la nota */
      delay(125);
}
```

Los otros sensores GP2Yxxx son similares al GP2Y0A02: la mayor diferencia entre ellos es el rango de distancias que pueden medir y el nivel de voltaje aportado según esta, pero su funcionamiento es muy parecido. Tanto en Adafruit como en Sparkfun podemos encontrar por ejemplo el sensor GP2Y0A21 (con código nº 164 y nº 242, respectivamente) que puede medir distancias entre 10 y 80 cm.

El sensor IS471F

Este sensor no es un detector de distancia propiamente sino simplemente de presencia; concretamente detecta la existencia o no de un obstáculo entre 1 cm y 15 cm. Funciona emitiendo un haz infrarrojo y comprobando si le llega rebotado. Si es así, el sensor generará una señal LOW (que podrá ser leída por una placa Arduino conectada a él convenientemente) y si no se detecta ningún objeto, el sensor generará una señal HIGH.

El sensor consta de cuatro pines, los cuales son (si observamos su cara plana de frente, de izquierda a derecha): alimentación (5 V), detección de datos (a conectar a un pin de entrada digital de nuestra placa Arduino), tierra y emisión de señal infrarroja (a este lo llamaremos pin “X”). Se recomienda conectar también un condensador (de 0,33 µF) de tipo “by-pass” entre ese pin “X” y el pin de tierra.

Lo que más sorprende de este sensor es que no es capaz de emitir por sí mismo ninguna señal infrarroja, por lo que para que funcione es necesario conectar al pin “X” del sensor el cátodo un diodo LED infrarrojo externo (preferiblemente de 940 nm). La emisión de este LED es convenientemente modulada a 38 KHz por un modulador interno que contiene el sensor, de manera que este sea relativamente inmune a las interferencias causadas por otro tipo de luz como la del sol (el rebote es a su vez demodulado por un demodulador interno). El ánodo de ese LED infrarrojo deberá conectarse a la fuente de alimentación apropiada, generalmente a través de un divisor de tensión. Si ese divisor de tensión lo convertimos en un potenciómetro, podremos regular dentro de unos límites la distancia hasta la que se puede detectar el objeto, ya que cuanto más baja sea la resistencia de ese potenciómetro más intensa será la luz emitida por el LED y, por tanto, mayor será esa distancia.

Los sensores QRD1114 y QRE1113

El sensor QRD1114 (código de producto 246 de Sparkfun) en realidad no es más que un emisor de infrarrojos y un fototransistor bajo el mismo encapsulado, por lo que el principio de funcionamiento es similar a los sensores infrarrojos analógicos ya vistos: cuanta más distancia tenga el obstáculo, menos voltaje de salida obtendremos del sensor. Su característica más destacable es su rango de distancias, ya que solo es capaz de detectar objetos situados entre 0 y 3 cm de él. En realidad, este componente no está pensado para medir distancias exactas, sino tan solo para comprobar la proximidad de objetos por debajo de esos 3 cm.

El código Arduino a utilizar con este sensor es idéntico al mostrado con el sensor LV-EZO: no se trata más que de obtener los valores de una entrada analógica. Sin embargo, las conexiones son algo más complejas: este sensor dispone de cuatro

patillas porque en realidad ya hemos comentado que se compone simplemente de un LED infrarrojo (ánodo y cátodo) y un fototransistor (colector y emisor). La patilla correspondiente al ánodo se ha de conectar a través de un divisor de tensión de 200 Ω al pin de alimentación de 5 V de nuestra placa Arduino, la patilla correspondiente al colector se ha de conectar a un pin de entrada analógica de nuestra placa Arduino y las otras dos patillas se han de conectar a la tierra común. Además, entre el ánodo y el colector se ha de conectar una resistencia de tipo "pull-up" cuyo valor puede oscilar entre 4,7 KΩ y 5,6 KΩ (según el que sea, los valores leídos por la placa Arduino cambiarán).

Este sensor también se puede utilizar además para detectar superficies blancas y negras (por lo tanto, podría ser utilizado en la construcción de robots seguidores de líneas) ya que las superficies blancas reflejan más luz que las negras, obteniéndose por tanto lecturas mayores. No obstante, existe un sensor más específico para realizar esta tarea concreta: el QRE1113.

El sensor QRE1113 es comercializado en forma de placa breakout por Sparkfun (producto nº 9453), la cual simplemente ofrece tres conectores: "VCC" (a enchufar a una fuente de 5 V), "GND" y "OUT" (a enchufar a una entrada analógica de nuestra placa Arduino). En realidad, este producto es la versión analógica de la placa breakout, ya que también se puede encontrar la versión digital. En la versión analógica, valores mayores para el voltaje leído significa que la luz emitida por el LED ha sido reflejada con mayor intensidad y por tanto que ha sido recibida mejor por el fototransistor (es decir, que se ha detectado una superficie clara).

SENSOR DE INCLINACIÓN



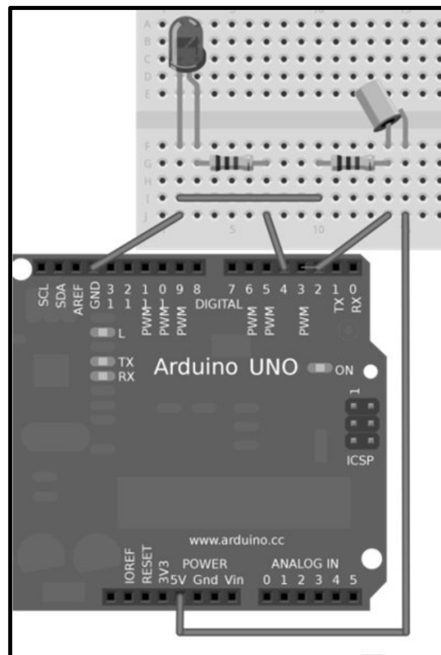
Los sensores de inclinación son pequeños, baratos, y fáciles de usar. Pueden trabajar con voltajes de hasta 24 V e intensidades de 5 mA. Constan de una cavidad y de una masa libre conductiva en su interior (como por ejemplo una bola metálica rodante); un extremo de la cavidad tiene dos polos conductivos de manera que cuando el sensor se orienta con este extremo hacia abajo, la masa rueda hacia los polos y los cierra. Por tanto, estos sensores actúan como interruptores, dejando o no pasar la corriente según la inclinación del circuito. Aunque no son tan precisos o flexibles como un completo acelerómetro, pueden detectar orientación o movimiento fácilmente.

Comprobar un sensor de inclinación es fácil: hay que poner el multímetro en modo de continuidad y conectar un cable cualquiera del multímetro (ya que los sensores de inclinación son dispositivos no polarizados) a cada borne del sensor. Seguidamente, hay que inclinarlo para determinar el ángulo en el cual el interruptor se abre y se cierra.

Si se conecta este sensor en serie a un LED (y su divisor de tensión preceptivo) y se alimenta el circuito, veremos cómo el LED se enciende o se apaga según la inclinación que le demos al diseño, tal como si estuviéramos utilizando un pulsador "invisible".

Si queremos usarlo con nuestra placa Arduino, debemos seguir la misma receta que empleamos cuando vimos los pulsadores: se pueden conectar utilizando una resistencia "pull-up" o "pull-down" (valores entre 200 Ω y 1 KΩ están bien).

Ejemplo 7.25: En el siguiente circuito mostramos un ejemplo básico, donde se puede apreciar que la lectura hecha por el sensor se recibe en un pin de entrada digital de la placa (hemos usado el nº 2) y se utiliza para controlar un LED conectado al pin digital de salida nº 4 (a través de su divisor de tensión), el cual se encenderá o no según el valor (HIGH o LOW) detectado por el sensor.



Y el código:

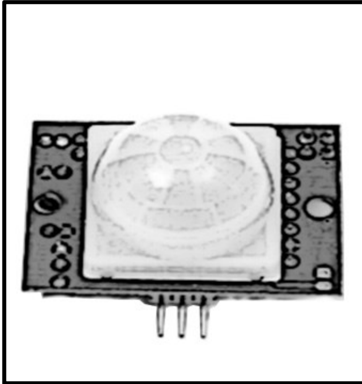
```
void setup(){
    pinMode(4,OUTPUT);
    pinMode(2, INPUT);
}
void loop(){
    digitalWrite(4,digitalRead(2));
}
```

Al sketch anterior se le podría añadir un control de “bouncing” como a los pulsadores, para aumentar la fiabilidad de las lecturas. Se deja como ejercicio.

SENSORES DE MOVIMIENTO

En este apartado hablaremos concretamente de los sensores “PIR” (del inglés “Pyroelectric passive InfraRed sensors”). La piroelectricidad es la capacidad que tienen ciertos materiales para generar un cierto voltaje cuando sufren un cambio de temperatura. Pero ojo, si su temperatura (sea alta o baja) se mantiene constante, ese voltaje poco a poco irá desapareciendo.

¿Y esto qué tiene que ver con el movimiento? Los sensores PIR básicamente se componen de dos sensores piroeléctricos de infrarrojos. Y todos los objetos emiten radiación infrarroja, estando además demostrado que cuanto más caliente está un objeto, más radiación de este tipo emite. Normalmente, ambos sensores detectarán la misma cantidad de radiación IR (la presente en el ambiente procedente de la habitación o del exterior), pero cuando un cuerpo caliente como un humano o un animal pasa a través del rango de detección, lo interceptará primero uno de los dos sensores, lo que causa un cambio diferencial positivo respecto el otro. Cuando el cuerpo caliente abandona el área de sensibilidad, ocurre lo contrario: es el segundo sensor el que intercepta el cuerpo y genera un cambio diferencial negativo. Estos pulsos son lo que en realidad el sensor detecta. Así pues, estos sensores son casi siempre utilizados para saber si un humano se ha movido dentro o fuera del (normalmente amplio) rango del sensor: alarmas de seguridad o luces de casa automáticas son un par de usos comunes para estos dispositivos.



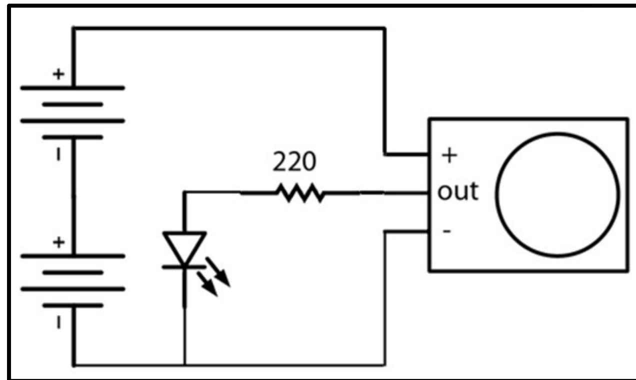
El sensor IR es básicamente un transistor FET con una ventana sensible a la radiación infrarroja en su cubierta protectora; cambios en el nivel de luz IR con una longitud de onda correspondiente al calor del cuerpo causan cambios en la resistencia fuente-drenador, que es lo que el circuito monitoriza. De todas formas, el gran truco de un sensor IR está en las lentes que incorpora: su función es condensar una gran área en una pequeña, proporcionando al sensor IR un rango de barrido mayor. De hecho, la calidad de las lentes es lo que básicamente diferencia un modelo de sensor PIR de otro, ya que la circuitería es bastante común a todos; cambiando una lente por otra se puede cambiar la amplitud y el patrón de sensibilidad del sensor.

Además de toda la circuitería ya comentada (sensores IR, lentes, etc.), un sensor PIR incorpora siempre un chip que sirve para leer la salida de los sensores IR y procesarla de tal manera que finalmente se emita un pulso digital al exterior (que es lo que nuestra placa Arduino recibirá).

La mayoría de sensores PIR vienen en plaquitas con 3 pines de conexión a un lado o al fondo: alimentación, tierra y señal de datos. El orden de estos tres pines puede variar según el modelo, así que hay que consultarlo en el datasheet (aunque la mayoría de veces cada pin ya tiene serigrafiada en la propia plaquita su función). La alimentación usualmente es de 3-5 V DC pero puede llegar a ser de 12 V, así que con una fuente de 5 V-9 V ya funcionan perfectamente.

Otras características comunes a la mayoría de modelos es que a través de su pin de datos emiten un pulso HIGH (3,3 V) cuando se detecta movimiento y emiten un pulso LOW cuando no. La longitud de los pulsos se determinan por los resistores y condensadores presentes en la PCB y difieren de sensor a sensor. Su rango de sensibilidad suele ser hasta una distancia de 6 metros y un ángulo de 110° en horizontal y 70° en vertical. La mayoría de modelos integran el chip BIS0001 para el control de la circuitería interna y el sensor IR RE200B; las lentes pueden ser muy variadas. Modelos concretos de sensores PIR son por ejemplo el producto nº 189 de Adafruit, el producto nº 8630 de Sparkfun o el producto 555-28027 de Parallax, entre otros.

Para probar el sensor PIR, podemos diseñar un circuito como el siguiente. Hay que tener en cuenta que el orden de los pines –alimentación, señal, tierra– es diferente según el modelo de sensor utilizado. Concretamente, el mostrado en el diagrama corresponde al modelo PIR de Adafruit, pero el orden de pines en el sensor 555-28027 es otro (concretamente, si lo miramos de frente, de izquierda a derecha tenemos los pines de señal, alimentación y tierra) y en el sensor de Sparkfun también (concretamente, si lo miramos de frente y de izquierda a derecha, tenemos los pines de señal, tierra, alimentación). En el sensor de Sparkfun es necesario además conectar también una resistencia "pull-up" de 10 KΩ entre su pin de alimentación y su pin de datos para que funcione correctamente; esto implica que, a diferencia de los anteriores, el sensor de Sparkfun envía una señal HIGH cuando no detecta movimiento y una señal LOW cuando sí.



La idea del circuito mostrado es que cuando el sensor PIR (suponiendo el modelo de Adafruit, que no usa resistencia pull-up externa) detecte el movimiento, el pin de datos enviará un pulso HIGH de 3,3 V y por tanto, iluminará el LED. Cuando el LED se apague, se puede probar de pasar la mano por delante, o directamente todo el cuerpo.

Hay que tener en cuenta que cuando se conectan las baterías, se ha de esperar entre medio minuto y un minuto a que el PIR se estabilice y comience a emitir datos fiables, así que al principio el LED puede parpadear un poco.

También hay que aclarar que el comportamiento del sensor PIR de Adafruit tiene la posibilidad de funcionar en dos modos según la posición de un jumper situado en el dorso de la placa. Si está colocado en la posición "L", el LED parpadeará a un ritmo de segundo a segundo mientras detecte movimiento, y si está colocado en la posición "H", el LED permanecerá encendido mientras detecte movimiento. El

sentido del modo "L" está en que (por poner un ejemplo) el sensor PIR puede estar conectado a algún otro dispositivo que se active cuando detecte un cierto número de parpadeos del LED.

Por su parte, si usamos el sensor PIR de Parallax, podemos especificar hasta qué distancia queremos abarcar (en dirección frontal al sensor) para detectar movimiento. Esto se hace mediante un jumper: si se coloca en la posición "S", se detectarán movimientos que ocurran dentro de una distancia de 5 metros al sensor; si se coloca en la posición "L", se detectarán movimientos que ocurran dentro de una distancia de 9 metros (en este modo, no obstante, pueden ocurrir falsos positivos).

Ejemplo 7.26: Conectar un sensor PIR (cualquiera de los mencionados) a nuestra placa Arduino es fácil: solamente hay que conectar el pin de datos del sensor a un pin-hembra de entrada digital, por ejemplo el nº 2 (además de a la alimentación de 5 V y tierra, obviamente). El sketch siguiente simplemente notifica por el canal serie cuándo se ha detectado movimiento.

```
int disparo= LOW;  //No hay movimiento al principio
int lectura = 0;
void setup() {
       pinMode(13, OUTPUT);
       pinMode(2, INPUT);
       Serial.begin(9600);
}
void loop(){
  lectura = digitalRead(2);
  if (lectura == HIGH) {
    digitalWrite(13, HIGH);
    if (disparo == LOW) {
      Serial.println("Movimiento detectado");
      //Solo queremos imprimir el mensaje una sola vez
      disparo = HIGH;
    }
  } else {
    digitalWrite(13, LOW);
    if (disparo == HIGH){
      Serial.println("Movimiento finalizado");
      //Solo queremos imprimir el mensaje una sola vez
      disparo = LOW;
    }
  }
}
```

El sensor ePIR

En Sparkfun (entre otros sitios) se puede adquirir con código de producto 9587 un sensor algo diferente llamado “ePIR”, del fabricante Zilog. La diferencia más importante entre este componente y los sensores PIR vistos anteriormente está en que el primero incluye además un microcontrolador propio dentro de su encapsulado. Esto permite una mayor flexibilidad a la hora de controlar el sensor y de gestionar los datos obtenidos.

Concretamente, se puede establecer comunicación con este componente de dos formas diferentes: en “modo hardware” y en “modo serie”. En el “modo hardware”, se puede ajustar la sensibilidad del sensor (es decir, a partir de qué valor detectado se considera movimiento) o el retardo (es decir, cuánto tiempo se esperará el sensor después de la detección de movimiento para volver a continuar detectando otro nuevo), entre otros parámetros.

El “modo serie”, por su parte, ofrece todas las funcionalidades del “modo hardware” pero además permite configuraciones más avanzadas mediante el envío de comandos específicos (como por ejemplo detectar movimiento en solo una dirección, ampliar el rango de detección de 3 m x 3 m a 5 m x 5 m y más). Por ejemplo, para forzar la detección de movimiento, deberemos enviar el comando “a”, el cual sirve para obtener la lectura del sensor en forma de carácter ASCII ‘Y’ (si hay movimiento) o ‘N’ (si no). De todas formas, para conocer todos los comandos disponibles y sus posibles usos, recomiendo consultar el datasheet del sensor.

Para seleccionar el “modo hardware”, en el momento de arrancar el sensor (o al salir de su modo de bajo consumo) se debe proporcionar una tensión menor de 1,8 V a su pin nº 4. Además, el valor concreto de esta tensión determina la sensibilidad del sensor, donde 0 V corresponde a la mayor sensibilidad y 1,8 V a la menor. Por lo tanto, si este pin se conecta directamente a tierra tendremos el “modo hardware” ya activado con la sensibilidad máxima posible. Si lo que se desea es regular dicha sensibilidad a mano, una opción sería conectar este pin a la patilla central de un potenciómetro (los extremos del potenciómetro en este caso deberían ir conectados a tierra y alimentación, respectivamente), utilizando los divisores de tensión pertinentes para obtener el rango 0-1,8 V deseado.

Para seleccionar el “modo serie”, en el momento de arrancar el sensor (o al salir de su modo de bajo consumo) se debe proporcionar una tensión mayor de 2,5 V a su pin nº 4. Una manera de conseguir esto es conectar una resistencia “pull-up” (generalmente de 100 KΩ) entre este pin nº 4 y la fuente de alimentación (que, atención, ha de ser de 3,3 V).

Los demás pines del sensor tienen una utilidad diferente según el modo de trabajo configurado. En el “modo hardware”, las conexiones necesarias son:

Pin nº 1: este pin se ha de conectar a tierra.

Pin nº 2: este pin se ha de conectar a la fuente de alimentación. esta ha de ser de entre 2,7 y 3,6 V, por lo que el pin “3V3” de la placa Arduino es ideal.

Pin nº 3: este pin sirve para especificar el retardo del sensor (recordemos que es el tiempo que el sensor esperará desde que ha detectado movimiento hasta ponerse otra vez a detectar). En realidad, no es más que una entrada analógica que puede recibir desde 0V (correspondiente a un retardo de 2 s) hasta 2 V (correspondiente a un retardo de 15 m), por lo que se podría conectar por ejemplo a un potenciómetro para regular la tensión aplicada a mano. Si se conecta a tierra, el retraso será fijo de 2 s.

Pin nº 4: este pin sirve para seleccionar el tipo de modo de trabajo (hardware o serie) que se quiere utilizar. El procedimiento concreto se ha explicado en los párrafos anteriores.

Pin nº 5: este pin se ha de conectar a una entrada digital de nuestra placa Arduino. Mediante este pin se recibe un valor LOW si se detecta la existencia de movimiento, o un valor HIGH si no.

Pin nº 6: este pin no es más que una entrada analógica que debería recibir un valor de tensión proporcional a la cantidad de luz existente en el ambiente. La idea es activar la detección de movimiento solamente en entornos de poca iluminación (en horario nocturno, por ejemplo). Concretamente, cuando la tensión aplicada a este pin es menor de 1 V, la detección de movimiento está desactivada. Por eso, normalmente, este pin se conecta a un fotorresistor, aunque también se puede utilizar un potenciómetro para tener un control más directo. Si esta funcionalidad no se desea, bastará con conectar este pin directamente a la fuente de alimentación de 3,3 V.

Pin nº 7: este pin se ha de conectar a una salida digital de nuestra placa Arduino. Si se envía a este pin una señal LOW, el sensor pasará a un estado de bajo consumo (“sleep mode”) y se inactivará. Este estado es útil cuando el sensor no se está utilizando. Cuando recibe una señal HIGH, vuelve a activarse.

Pin nº 8: este pin se ha de conectar a tierra.

En el caso de querer utilizar el “modo serie”, las conexiones necesarias son:

Pin nº 1: mismo uso que en el “modo hardware”.

Pin nº 2: mismo uso que en el “modo hardware”.

Pin nº 3: este pin se ha de conectar al pin TX de la placa Arduino (o uno simulado con la librería SoftwareSerial). Sirve para recibir los comandos provenientes de esta.

Pin nº 4: además de servir para seleccionar el tipo de modo de trabajo (hardware o serie) que se quiere utilizar (el procedimiento concreto se ha explicado en párrafos anteriores), también sirve recibir las lecturas realizadas por el sensor, por lo que además se deberá conectar al pin RX de nuestra placa Arduino (o uno simulado con la librería SoftwareSerial).

Pin nº 5: este pin se ha de conectar al pin RESET de la placa Arduino.

Pin nº 6: mismo uso que en el "modo hardware".

Pin nº 7: mismo uso que en el "modo hardware".

Pin nº 8: mismo uso que en el "modo hardware".

Ejemplo 7.27: A continuación, mostramos un código de ejemplo de uso del "modo hardware", en el cual un LED (conectado a la salida digital nº 12 de nuestra placa Arduino) se ilumina cada vez que se detecta movimiento:

```
int sleepModePin = 4;     //Donde se conecta el pin n° 7 del sensor
int motionDetectPin = 2; //Donde se conecta el pin n° 5 del sensor
int lectura;
void setup() {
   pinMode(12, OUTPUT);
   pinMode(sleepModePin, OUTPUT);
   pinMode(motionDetectPin, INPUT);
   /*El pin "sleepModePin" ha de recibir una señal HIGH
   para activar la detección de movimiento */
   digitalWrite(sleepModePin, HIGH);
}
void loop() {
  lectura = digitalRead(motionDetectPin);
  if(lectura == LOW) {    //Se detecta movimiento
    digitalWrite(12, HIGH);
  } else {                //No se detecta movimiento
    digitalWrite(12, LOW);
  }
  //Me espero dos segundos para volver a detectar movimiento
  delay(2000);
}
```

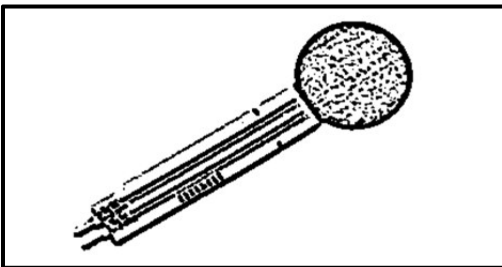
Ejemplo 7.28: A continuación, mostramos un código de ejemplo de uso del "modo serie", en el cual hemos sustituido el LED por un mensaje leído por el canal serie indicando si se ha detectado movimiento o no:

```
#include <SoftwareSerial.h>
int sleepModePin = 4;     //Donde se conecta el pin n° 7 del sensor
```

```
int lectura;
/*El pin 4 es el RX de la placa (conectado al n° 4 del sensor)
El pin 3 es el TX de la placa (conectado al n° 3 del sensor)*/
SoftwareSerial ePir = SoftwareSerial(4,3);
void setup() {
       pinMode(sleepModePin, OUTPUT);
       digitalWrite(sleepModePin, HIGH);
       /*La comunicación con el ePIR en modo serie
       ha de ser siempre a 9600 bits/s.*/
       ePir.begin(9600);
       Serial.begin(9600);
}
void loop() {
      //Ordeno la detección de movimiento
      ePir.print("a");
      //Mientras no se reciba respuesta del sensor, no hago nada
      while(!ePir.available()) {;}
      /*Muestro la respuesta del sensor:
      'Y' si hay movimiento y 'N' si no */
      Serial.print(ePir.read(););
      //Me espero dos segundos para volver a detectar movimiento
      delay(2000);
}
```

SENSORES DE CONTACTO

Sensores de fuerza



Estos sensores (también llamados FSRs, del inglés, "Force Sensitive Resistor") permiten detectar fuerza. Son básicamente un resistor que cambia su resistencia dependiendo de la fuerza a la que es sometido (concretamente, su resistencia disminuye a mayor fuerza recibida). Estos sensores son muy baratos y fáciles de usar pero no demasiado precisos: una misma medida puede variar entre un sensor y otro hasta un 10%. Así que lo que uno puede esperar de un FSR es conseguir medir rangos de respuesta; es decir: aunque los FSRs sirven para detectar peso, son mala elección para detectar la cantidad exacta de este. Sin

embargo, para la mayoría de aplicaciones sensibles al tacto, del tipo “esto ha sido apretado cierta cantidad” son una solución aceptable y económica.

Los FSRs están compuestos por una zona “activa” (de forma generalmente circular o cuadrada y de diferentes tamaños según el modelo), y dos terminales que, al ser este dispositivo una resistencia, no están polarizados. Normalmente, pueden soportar rangos de fuerzas de 0 a 100 newtons y el rango de resistencias que ofrecen van desde resistencia infinita cuando no detectan fuerza hasta aproximadamente 200 ohmios a máxima fuerza. En el datasheet del modelo concreto de FSR deberemos encontrar siempre cómo es esa relación “fuerza aplicada -> resistencia obtenida”, la cual no es exactamente lineal (ya que a pequeñas fuerzas hay una variación muy grande de la resistencia, y a fuerzas mayores la variación ya es menor).

Para comprobar su funcionamiento se puede utilizar un multímetro en modo de medida de resistencia. Apretando la zona sensible del FSR se deberá observar los cambios de resistencia.

Ejemplo 7.29: Si queremos utilizar este componente en un circuito con Arduino, debemos conectar un terminal a la alimentación y el otro a una resistencia “pull-down” (de 10 KΩ por ejemplo), la cual deberá ir a tierra. Además, un punto entre la resistencia “pull-down” (fija) y la resistencia FSR (variable) lo debemos conectar a una entrada analógica de la placa Arduino. Es el mismo montaje que ya vimos cuando hablamos de los LDRs y de los termistores, de hecho. La idea también es la misma que en los casos anteriores: leer el voltaje que hay en ese punto, que incrementa a medida que, por simple Ley de Ohm, la resistencia del FSR disminuye (es decir, a medida que se le aplica más fuerza).

A continuación, se presenta un sketch muy parecido a otros vistos anteriormente, el cual enciende de forma progresiva un LED según vayamos apretando más un FSR. En este ejemplo, ese LED está conectado (a través de su inseparable divisor de tensión) al pin de salida PWM nº 11. El FSR está conectado al pin de entrada analógica nº 0.

```
int lectura;
int brillo;
void setup() {
	pinMode(11, OUTPUT);
}
void loop() {
	lectura = analogRead(0); //Cuanta más fuerza, más voltaje leído
```



```
	brillo = map(lectura, 0, 1023, 0, 255);
	analogWrite(11, brillo); //A más fuerza, más brillo
	delay(100);
}
```

De forma muy similar podríamos realizar otro circuito muy interesante, consistente en un servomotor controlado mediante un FSR, de tal forma que el ángulo de giro de aquel fuera proporcional a la fuerza ejercida sobre este. El truco estaría en usar *map()* para mapear el rango de valores 0-1023 leídos por *analogRead()* al rango de valores 0-179, que es el admitido por *miservo.write()* (función que tendría que sustituir al *analogWrite()* del código anterior). Se deja como ejercicio.

Si lo que nosotros queremos saber en realidad es el valor concreto de la resistencia del FSR, podemos utilizar en nuestros sketches la siguiente fórmula: $R_{fsr} = (R_{pull} \cdot 1023/V_{convertido}) - R_{pull}$, donde $V_{convertido}$ es el valor leído por la entrada analógica de la placa Arduino una vez transformado por el conversor analógico/digital, R_{pull} es el valor de la resistencia "pull-down" (de valor fijo) y R_{fsr} es el valor de la resistencia del FSR que deseamos conocer. Esta fórmula es idéntica a las obtenidas anteriormente en el estudio de los fotorresistores y los termistores, ya que el razonamiento para obtenerla es el mismo.

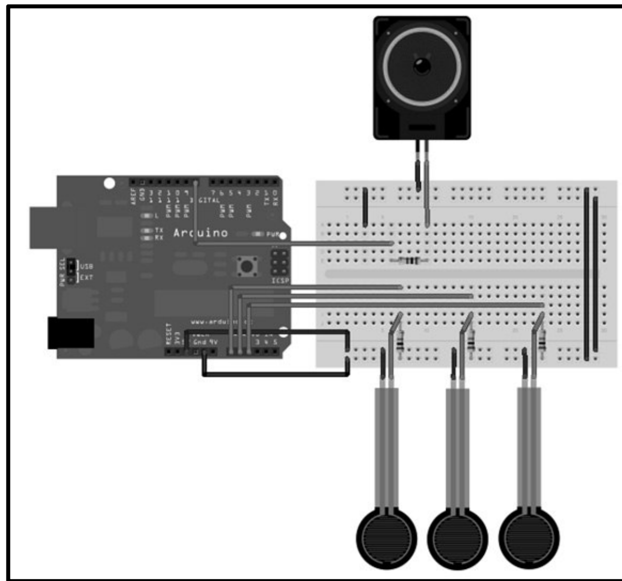
Conociendo R_{fsr} podríamos deducir la cantidad de fuerza recibida por el sensor, pero desgraciadamente no existe una fórmula analítica que relacione ambas magnitudes, por lo que no tenemos más remedio que consultar la gráfica del datasheet para conocer la relación exacta entre el valor de R_{fsr} calculado y el valor de la fuerza correspondiente.

Ejemplo 7.30: El siguiente sketch hace uso de la fórmula mencionada en el párrafo anterior para obtener la resistencia del FSR. El circuito es el mismo que el del sketch anterior, pero sin LED: usaremos el "Serial monitor" para leer los diferentes datos obtenidos.

```
int lectura;
const int rpull = 10000 //La resistencia pull-down es de 10KΩ
unsigned long fsr;  //Pueden ser valores muy altos
unsigned long cond; //Pueden ser valores muy altos
long fuerza;
void setup() {
	Serial.begin(9600);
```

```
}
void loop() {
      lectura = analogRead(0);
      if (lectura == 0) {
           Serial.println("No hay fuerza");
      } else {
           fsr=(rpull*1023/lectura) - rpull;
           Serial.print("Resistencia: ");
           Serial.println(fsr);
/*La gráfica del datasheet muestra la fuerza en función no de la
resistencia, sino de su inversa, la conductancia, medida en
microMhos.Por eso se ha de invertir convenientemente el valor de
fsr*/
           cond = 1000000 / fsr;
/*Y ahora usamos los valores de la gráfica para aproximar el valor
estimado de la fuerza. Dependiendo del modelo concreto de FSR los
valores en las condiciones de los "ifs" deberán ser diferentes */
           if (cond <= 1000) {
               fuerza = cond / 80;
               Serial.print("Fuerza en Néwtones: ");
               Serial.println(fuerza);
           } else {
               fuerza = (cond - 1000) / 30;
               Serial.print("Fuerza en Néwtones: ");
               Serial.println(fuerza);
         }
    }
    delay(100);
}
```

Ejemplo 7.31: Un ejemplo curioso de aplicación de FSRs (o de hecho, de cualquier otro sensor analógico) es el siguiente circuito. En él tenemos 3 FSRs conectados en paralelo a los pines de entrada analógica de la placa Arduino nº 1, 2 y 3 y a tierra a través de una resistencia "pull-down" de 10 KΩ. Además, tenemos un zumbador o altavoz conectado al pin digital de salida nº 8 (a través de un divisor de tensión de 100 ohmios).

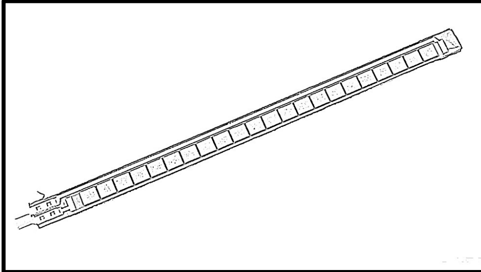


El código lo que hace es leer las lecturas de los tres sensores, cada uno de los cuales corresponde a una nota musical dentro de un array de notas. Si alguno de estos sensores se presiona más allá de un umbral, la nota correspondiente suena. Hemos implementado pues, un simple piano electrónico.

```
//Más allá del umbral sonará la nota
const int umbralminimo = 10;
//Cada nota corresponde a un sensor
int notas[] = { 220, 440, 880 };
void setup(){}
void loop() {
    int i;
    int lectura;
    //Voy recorriendo los sensores uno tras otro
    for (i = 0; i < 3; i++) {
         lectura = analogRead(i);
         //Si se presiona lo suficiente
         if (lectura > umbralminimo) {
               tone(8, notas[i], 20);
         }
    }
}
```

En Adafruit distribuyen un modelo de FSR con número de producto 166 y en Sparkfun venden varios con diferentes características: el nº 9375 y el nº 9376, pero todos son fabricados por Interlink (http://www.interlinkelectronics.com).

Sensores de flexión



Unos sensores parecidos a los FSR son los sensores de flexión (en inglés llamados "flex sensors" o "bend sensors"). Estos sensores están compuestos por una tira resistiva flexible solo en una dirección. Su resistencia cambia según cuánto sea arqueada: si están en equilibrio (es decir, sin combarse) su resistencia es mínima y cuanto más se flexiona más resistencia ofrece.

Al igual que las FSR, tienen dos terminales: uno podemos conectarlo dentro de nuestros circuitos a la fuente de alimentación (preferiblemente a través de un divisor de tensión) y el otro a una resistencia "pull-down" que va a tierra (también se podría usar la configuración alternativa usando "pull-up"s). En el punto donde se conecta el sensor a la resistencia "pull-down" se debe conectar una entrada analógica de la placa Arduino para leer el voltaje resultante en ese punto. Como ya sabemos, ese valor depende del valor de la resistencia del sensor, por lo que nos servirá para saber cuánto está flexionado.

Ejemplo 7.32: El siguiente sketch muestra un ejemplo muy básico de uso:

```
void setup(){
    Serial.begin(9600);
}
void loop() {
  int sensor, grados;
    sensor = analogRead(0);
/*Convierto el valor leído a grados de flexión. Los dos primeros
números de map() (768 y 853) son los valores leídos cuando el sensor
está completamente recto y cuando tiene una curvatura de 90 grados,
respectivamente. Estos valores los podemos haber consultado en el
datasheet o bien haberlos calibrado anteriormente. Los dos siguientes
números de map() son los grados a los que queremos mapear (0 grados y
ángulo recto)/*
  grados = map(sensor, 768, 853, 0, 90);
```



```
  Serial.print("Los grados de flexión son: ");
  Serial.println(grados,DEC);
  delay(100);
}
```

Sparkfun distribuye varios modelos de diferente longitud: el producto nº 10264 y el 8606 son dos ejemplos. Adafruit solo distribuye el segundo, con código 182. Además de la longitud, otras características importantes a tener en cuenta son su anchura y su peso, ya que muchas veces estos sensores se utilizan en proyectos textiles.

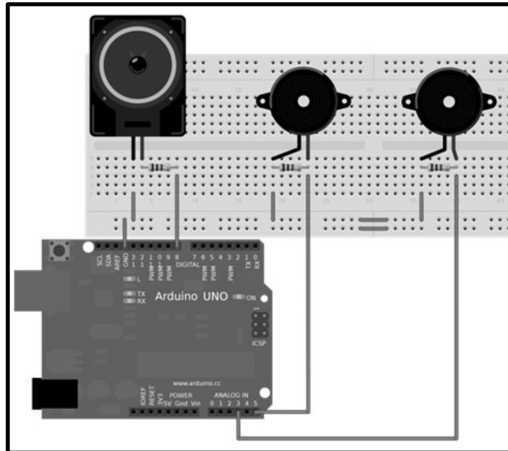
No confundir los sensores de flexión con los llamados sensores "Flexiforce", que es una marca registrada de Tekscan (http://www.tekscan.com). Los sensores "Flexiforce" son sensores FSR y Sparkfun ofrece unos cuantos que soportan diferentes rangos de fuerza (productos nº 11207, 8685, 8712 o 8713, entre otros).

Sensores de golpes

Debido a su constitución eléctrica interna, los zumbadores también pueden utilizarse, además de como emisores de sonidos, como sensores de golpes. El mecanismo es justo a la inversa del convencional: los golpes (suaves) recibidos por el zumbador se traducen en vibración de su lámina interna, la cual genera una serie de pulsos eléctricos que pueden ser leídos por una placa Arduino. De esta manera, podemos diseñar circuitos que respondan al tacto y que distingan incluso la presión ejercida. Como el zumbador es un dispositivo analógico, según lo fuerte que se golpee, la señal leída por la placa Arduino será de menor o mayor intensidad.

Ejemplo 7.33: Sabiendo esto, podemos diseñar el circuito de ejemplo mostrado a continuación, donde tenemos un altavoz y dos zumbadores. Tal como se puede ver, el altavoz está conectado a tierra y a la salida digital nº 8 (a través de una resistencia en serie de 100 ohmios), los terminales positivos de los zumbadores (si estos son polarizados) están conectados a las entradas analógicas nº 3 y nº 5, y los terminales negativos de los zumbadores (si estos son polarizados) a tierra. Además, cada zumbador está conectado en paralelo a una resistencia (de un valor recomendable de 1 MΩ), cuya función es actuar como resistencia "pull-down", manteniendo la entrada analógica a 0 V mientras el zumbador no sea presionado.

Si no tenemos ningún zumbador a mano, podemos conseguir el mismo efecto adquiriendo una lámina piezoeléctrica tal como el producto nº 10293 de Sparkfun, que no es más que un zumbador sin su recubrimiento, o adquiriendo el producto nº 10772, consistente en un kit de cuatro láminas como la anterior más varias resistencias de 1 MΩ. También podemos adquirir el "Sound & Buzzer Module" de Freetronics, que no es más que un zumbador incorporado a una cómoda plaquita breakout.



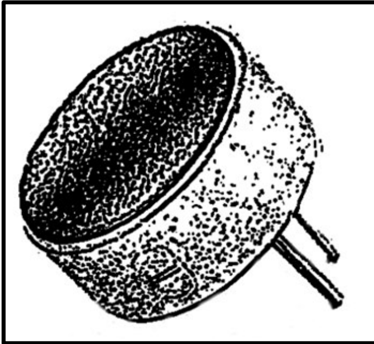
A partir de este circuito, podemos escribir un código como el siguiente. La idea es que según el zumbador que pulsemos, sonará por el altavoz una nota u otra. Pero además, cuanto más fuerte apretemos un zumbador, su nota sonará más tiempo. Esto es posible porque podemos conocer la presión ejercida sobre el zumbador: estos alcanzan una amplitud de vibración que depende de la magnitud del golpe recibido: a más presión, más amplitud. Y la amplitud se traduce proporcionalmente en voltaje, voltaje detectado por las entradas analógicas de la placa Arduino. Como precisamente el zumbador es un dispositivo analógico, la amplitud de su vibración al sufrir un golpe (es decir, el nivel de voltaje recibido) aumentará poco a poco, llegará a su máximo y volverá a decrecer hasta cesar. Por tanto, el truco para detectar la magnitud del golpe no está realmente en detectar el voltaje máximo obtenido (que puede ser un dato no muy preciso) sino en comprobar durante cuánto tiempo se mantiene el voltaje recibido por encima de un determinado umbral (decidido por nosotros): cuanto más tiempo transcurra entre la primera y última vez que se detecta un valor de voltaje, mayor de ese umbral, mayor habrá sido la magnitud del golpe.

```
int umbral = 100; //El zumbador se considera pulsado sobre ese umbral
int lectura1 = 0; //Lectura obtenida de un zumbador
```



```
int lectura2 = 0; //Lectura obtenida del otro zumbador
int tiempo = 0; //Tiempo en que las lecturas son superiores al umbral
int do = 1915;    //Semiperíodo de la onda de la nota "do"
int re = 1700;    //Semiperíodo de la onda de la nota "re"
void setup(){
        pinMode(8, OUTPUT);
}
void loop(){
        //Se detecta si está presionado un zumbador
        lectura1= analogRead(3);
        if (lectura1 > umbral) {
                tiempo=0;
/*Mientras se detecte que la presión continúa, se sigue leyendo la
entrada analógica para ver si aún se lee un valor por encima del
umbral. El tiempo transcurrido hasta que se lea un valor menor al
umbral marcará la duración de la nota a escuchar*/
                while (lectura1 > umbral) { tiempo++; }
/*Para evitar posibles ruidos no deseados y considerar el golpe
válido, se establece un tiempo mínimo de presión */
                if (tiempo > 100) { sonido(do, tiempo); }
        }
        //Se detecta si está presionado el otro zumbador
        lectura2= analogRead(5);
        if (lectura2 > umbral) {
                tiempo=0;
                while (lectura2 > umbral) { tiempo++; }
                if (tiempo > 100) { sonido(re, tiempo); }
        }
}
void sonido(int nota, int tiempo ) {
         unsigned long duracion;
         duracion = micros() + (35000 * tiempo);
/*Empiezo a contar desde el momento actual y añado un valor
arbitrario multiplicado por el valor de "tiempo"; así, cuanto más
fuerte se pulse el zumbador, más durará */
        //Mientras no se llegue al final
        while(micros() < duracion){
            digitalWrite(8, 255);
            delayMicroseconds(nota);
            digitalWrite(8, 0);
            delayMicroseconds(nota);
        }
}
```

SENSORES DE SONIDO



En realidad, un sensor de sonido no es más que un sensor de presión que convierte las ondas de presión de aire (las ondas sonoras) en señales eléctricas de tipo analógico; es decir, un micrófono. Existen muchos tipos de micrófonos según el mecanismo físico que utilizan para realizar esa conversión: los de tipo "inductivo" (también llamados "dinámicos"), los "de condensador", los piezoeléctricos, etc. Dependiendo del tipo, unos tendrán una mejor respuesta a un rango determinado de frecuencias de sonido que otros (es decir, que serán más "fieles" a la onda original), unos tendrán una mayor sensibilidad que otros (es decir, que ya generarán un determinado voltaje a menores variaciones de volumen detectadas), unos comenzarán a distorsionar a menores volúmenes que otros (es decir, que ofrecerán una THD menor para un determinado voltaje), unos serán más resistentes y duraderos que otros, etc.

No obstante, en nuestros proyectos con placas Arduino UNO la variedad de micrófonos a elegir se reduce drásticamente. Arduino UNO no es una plataforma pensada para el procesamiento de audio: ya hemos visto que (aunque existen proyectos destacables en el ámbito de la síntesis) la generación y emisión de sonido es ciertamente limitada. Y lo mismo ocurre con la recepción de sonido: para empezar, los pines-hembra de la placa no son capaces de recibir corriente AC (que es lo que son las señales de audio). Además, el conversor analógico-digital tarda como mínimo 100 microsegundos en realizar una lectura de una entrada, por lo que la máxima frecuencia de muestreo posible es de 10 KHz (es decir, una calidad relativamente baja). Además, el procesamiento de una señal acústica (compuesta en realidad de un conjunto de múltiples señales analógicas de diferentes frecuencias y amplitudes) es mucho más complejo de lo que el ATmega328P y su limitada memoria es capaz de realizar con solvencia.

Por eso, los micrófonos que se utilizan junto con las placas Arduino UNO en la mayoría de los casos son utilizados solamente como simples detectores de presencia y/o volumen de sonido (o como mucho, conectándolos a chips especiales como el MSGEQ7 de Mixed Signal Integration, podrían ser usados como detectores de frecuencias de sonido, pero esta posibilidad no la trataremos).

Concretamente, nosotros utilizaremos una variante de micrófono de tipo condensador llamado "micrófono electret". Este tipo de micrófono es

omnidireccional (es decir, detecta el sonido de todas direcciones y no solo el proveniente de un punto en particular) y por lo general tienen una sensibilidad buena (es decir, que el voltaje generado varía bastante acordemente según lo que varíe la intensidad del sonido). Además, son baratos y de un tamaño reducido. Tienen mejor respuesta a frecuencias de rango medio-alto, por lo que son mejores para comunicaciones de voz que para música de sección rítmica importante, por ejemplo. En muchos dispositivos domésticos como teléfonos móviles, computadores o auriculares los micrófonos que vienen incorporados son de ese tipo.

Sea cual sea la aplicación práctica que le demos a un micrófono, en todo caso su uso implica necesariamente el uso de un pre-amplificador. Esto es debido a que la señal generada por un micrófono tiene una amplitud demasiado pequeña (generalmente entre 0 y 100 milivoltios) para poder ser aprovechada por nuestra placa Arduino. Por tanto, no podremos conectar un micrófono directamente a una entrada analógica de nuestra placa Arduino tal como hemos venido haciendo con otros sensores analógicos, sino que deberemos incluir un pre-amplificador entre el micrófono y la placa Arduino. En este sentido, hay que distinguir entre pre-amplificación (conversión de la señal generada por el micrófono para llevarla a un nivel usable por el circuito (en este caso, la placa Arduino) y amplificación (conversión de esa señal usable internamente en nuestro circuito a un nivel lo suficientemente audible para poder emitirlo a través de altavoces).

De hecho, este proceso de pre-amplificación y amplificación también es necesario cuando el micrófono lo conectamos a un sistema de audio doméstico o profesional: todos los dispositivos intermediarios (desde los reproductores portátiles de ficheros “mp3” o similares hasta cadenas de alta fidelidad pasando por televisores, reproductores de DVD, mesas de mezclas, etc.) trabajan a unos niveles de tensión superior al aportado por cualquier micrófono (el llamado “nivel de línea”, diferente a su vez del utilizado por Arduino) por lo que la señal generada por este ha de ser pre-amplificada para poder ser utilizada por todos estos dispositivos de audio. Una vez esta señal eléctrica ya está a nivel de línea, para transformarla en sonido real y poderlo emitir por algún sistema de altavoces es necesario entonces amplificarla hasta alcanzar la potencia deseada.

Plaquitas breakout

Existen plaquitas breakout que incorporan un micrófono electret y un pre-amplificador todo en uno, de manera que podamos empezar a utilizar el kit completo micrófono+pre-amplificador al instante. Un ejemplo es el “Microphone Sound Input Module” de Freetronics. La plaquita consta de cuatro conectores: “VCC” (a conectar a

la alimentación de 5 V proporcionada por nuestra placa Arduino), "GND" (a conectar a tierra), "MIC" (salida analógica a conectar a una entrada analógica de nuestra placa Arduino) y "SPL" (otra salida analógica a conectar a otra entrada analógica de nuestra placa Arduino). En nuestros proyectos podremos utilizar las señales recibidas por ambos canales ("MIC" y "SPL") pero lo normal es conectar y utilizar solo uno de ellos, según lo que nos interese: el canal "MIC" proporciona una señal amplificada lo más fiel posible a la señal acústica, por lo que es más útil cuando se desea realizar un procesado del audio; en cambio, el canal "SPL" (de "Sound Pressure Level") simplemente ofrece un voltaje que es proporcional al volumen de sonido recibido, por lo que nos servirá para detectar fácilmente la existencia de sonido y su "cantidad", que es lo que generalmente desearemos.

Ejemplo 7.34: El siguiente sketch lee el valor SPL proveniente de esta plaquita cinco veces por segundo y muestra la lectura por el canal serie: a más voltaje leído, más volumen de sonido.

```
//La salida SPL está conectada al pin de entrada analógico 0
const byte splSensor = 0;
void setup() {
 Serial.begin(9600);
}
void loop() {
  Serial.println(analogRead(splSensor));
  delay(200);  //Evito la sobrecarga del canal serie
}
```

Una plaquita similar a la anterior pero que solo ofrece una salida de tipo "SPL" es el producto nº DFR0034 de DFRobot. Otra plaquita similar que también ofrece solamente una salida de tipo SPL es ZX-Sound de Inex Robotics.

Por otro lado, también existen plaquitas cuya salida es digital: si detectan un sonido superior a un umbral (generalmente definido a través de un potenciómetro), envían una señal HIGH a la entrada digital de la placa Arduino donde estén conectadas, y si el sonido detectado es inferior a ese umbral, envían una señal LOW. Estas plaquitas son útiles cuando no se desea monitorizar el volumen del entorno, sino tan solo determinados sonidos con un volumen superior al resto (una aplicación práctica es detectar colisiones en robots, por ejemplo). Un ejemplo de plaquita de este estilo es el producto nº 29132 de Parallax; tan solo consta de tres conectores (5 V, GND y señal) y mediante un potenciómetro que lleva incorporado podemos calibrar la sensibilidad del sensor, de manera que el sonido umbral que separa el envío de una señal HIGH (se detecta ruido) de una LOW (no se detecta ruido) se

amolde a nuestras necesidades. Otra plaquita muy similar es la llamada “Sound Sensor TTL output” de Cutedigi.

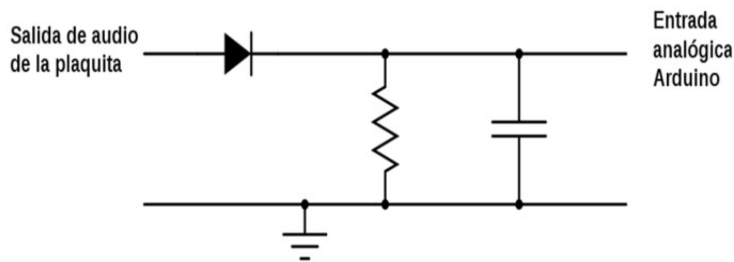
Ejemplo 7.35: A continuación, presentamos un código de ejemplo de uso muy sencillo para ambas plaquitas:

```
int lectura = 0;
void setup() { Serial.begin(9600); }
void loop() {
  //La plaquita está conectada a la entrada digital n° 2 de Arduino
  lectura = digitalRead(2);
  if (lectura == HIGH){ //Si se detecta un sonido más allá del umbral
    Serial.println("Sonido detectado");
    delay(100);
}
```

Una plaquita que tan solo ofrece una salida de tipo “MIC” (es decir, una señal eléctrica que reproduce el comportamiento de la onda acústica) es el producto nº 9964 de Sparkfun. Una vez conectada esta salida “MIC” a la entrada analógica de la placa Arduino (además del conector “VCC” a una fuente de 5 V y el conector “GND” a la tierra común), si observamos los valores recibidos por el “Serial monitor” veremos que cuando no se detecta sonido se mantienen estables alrededor del valor 512 y cuando hay ruido fluctúan por encima y por debajo de ese valor central (siendo mayor esta desviación –tanto por arriba como por abajo– cuanto mayor sea el volumen del ruido detectado). Por tanto, si quisiéramos obtener el volumen (es decir, utilizar esta plaquita como un simple sensor SPL), deberíamos utilizar la expresión `volume=abs(analogRead(0)-512);` (suponiendo que la entrada analógica utilizada es el pin-hembra nº 0). Pero ¿por qué este comportamiento?

Porque lo que hace esta plaquita es “trasladar” el valor de la salida correspondiente a 0 V (el valor central de la onda) a un valor central de 2,5 V y con él, el resto de valores en bloque sin alterar por tanto la forma de la onda. En otras palabras: añade un “colchón” de 2,5 V (DC) sobre el cual viaja la señal de audio (AC). Esto es lo que se llama tener una señal “descentrada” (o “biased”, en inglés). El objetivo es hacer que la placa Arduino pueda detectar tanto los valores positivos de la onda acústica como los negativos. Si no existiera el descentramiento de la señal, los valores negativos estarían realmente por debajo de 0 V y entonces la entrada analógica de la placa Arduino (que solo admite corriente DC y no AC) no podría detectarlos correctamente. Con la señal descentrada, el pico positivo podrá llegar a un máximo de 5 V (es decir, podrá tener una amplitud máxima de 2,5 V a partir de la señal base descentrada de 2,5 V) y el pico negativo a un mínimo de 0 V (es decir, podrá tener una amplitud máxima de -2,5 V a partir de la señal base descentrada de 2,5 V).

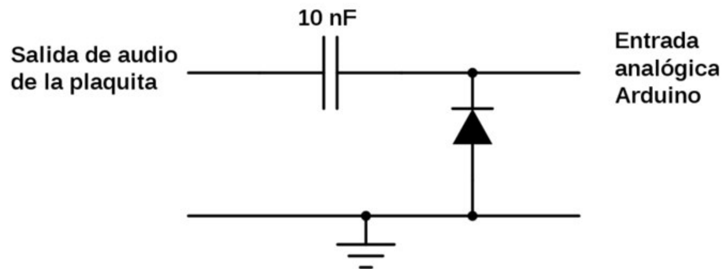
Otro truco que podemos utilizar con una salida de tipo "MIC" (tanto de la plaquita de Freetronics como la de Sparkfun) es distinguir si el sonido detectado es un sonido puntual (un portazo, por ejemplo) o bien un sonido continuado (como una conversación, por ejemplo). Debido a que la placa Arduino UNO es relativamente lenta a la hora de tomar muestras, es posible que no detecte todas las variaciones de volumen en tiempo real y se "deje por detectar" picos de sonido que aparezcan entremedio de dos lecturas. Para solventar esta cuestión, se puede acoplar a la salida "MIC" un circuito llamado genéricamente "detector de envolvente" (en inglés, "envelope detector"), mostrado a continuación.



El objetivo de este circuito es hacer llegar a la placa Arduino tan solo los valores extremos de cada pico positivo, uniéndolos entre sí de una forma suave, sin tener en cuenta por tanto toda la vibración de pico a pico. La figura de la señal recibida (la "envolvente") ayudará a la placa Arduino a "ir siguiendo" las variaciones de volumen: si la envolvente contiene picos estrechos habremos detectado un ruido brusco, y si es al contrario, estaremos detectando un sonido más o menos continuo. Pero ¿por qué este circuito funciona así?

En los picos positivos, el diodo deja pasar la señal, cargando el condensador hasta (casi) el valor del pico. En los picos negativos el diodo está polarizado inversamente, no permitiendo que fluya la corriente, por lo que el condensador reaccionará descargándose lentamente a través del resistor (la entrada de Arduino está diseñada para atraer una cantidad de corriente despreciable). Cuando vuelve a aparecer un pico positivo, el diodo vuelve a dejar pasar la señal y el condensador se vuelve a cargar rápidamente, y así todo el rato. La velocidad de descarga del condensador viene definida por el producto R·C (llamado "constante de tiempo"), y su valor dependerá del tipo de proyecto que deseemos realizar, pero un valor típico para empezar a probar es 0,1.

Otra cosa que podemos hacer con la salida “MIC” es convertirla en una salida “SPL”. En el caso de la placa Freetronics no nos será necesario porque ya tenemos los dos tipos de salidas, pero en el caso de la plaquita de Sparkfun, el truco está en acoplar a la salida “MIC” un circuito llamado genéricamente “fijador de nivel positivo” (en inglés, “positive unbiased clamper”), mostrado a continuación.



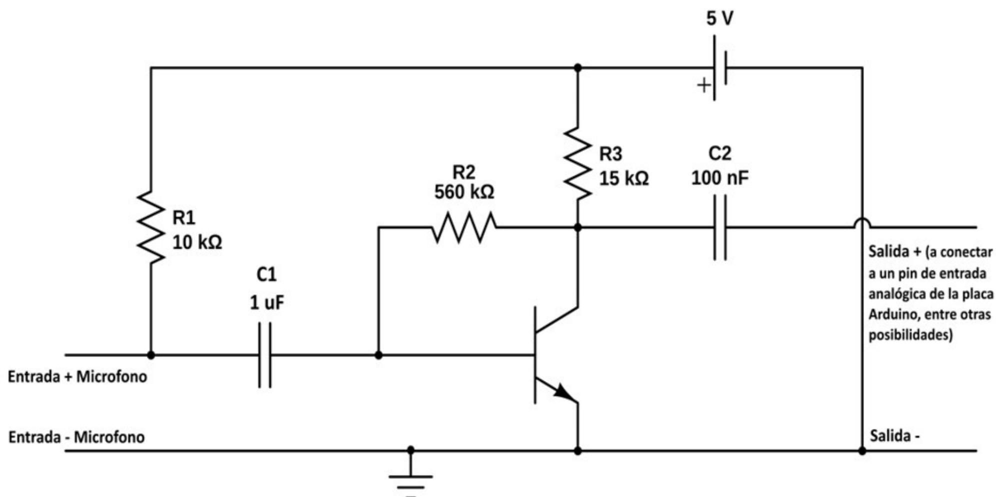
El objetivo del circuito anterior es descentrar la señal AC de tal forma que, sin modificar la forma de su onda, los extremos de los picos negativos tengan siempre justo un valor mínimo de 0 V. Si el volumen de sonido cambia, la longitud de los picos de la señal también, pero como lo que está fijado es que el valor mínimo sea siempre 0 V, lo que ocurrirá es que el valor central se deberá desplazar para cumplir esta condición, con lo que cambiará el valor extremo por el otro lado, por el pico positivo: cuando el volumen aumente, el valor extremo del pico positivo aumentará y cuando el volumen disminuya, el valor extremo del pico positivo disminuirá también. Y esto es lo que mediremos. Se recomienda que el diodo sea de tipo “germanio” (el 1N34A sería un buen ejemplo) y en este caso, se recomienda también alimentar la plaquita con 3,3 V en vez de 5 V. Si se hace así, los valores recibidos por la entrada analógica de la placa Arduino oscilarán entre 0 y 750/800 (a más volumen, mayor valor máximo recibido). Podremos entonces escribir un sketch que reaccionara a sonidos por encima de un umbral determinado, por ejemplo, simplemente observando los valores analógicos máximos recibidos.

Circuitos pre-amplificadores

Es posible que solo dispongamos de un micrófono electret en vez de toda una plaquita breakout. Si queremos conectarlo a nuestro circuito, lo primero que debemos saber es que los micrófonos electret (como el producto nº 8635 de Sparkfun, por ejemplo) deben ser alimentados. Son además dispositivos polarizados, en los cuales su terminal negativo suele estar marcado mediante una muesca o señal. En principio, este terminal negativo debería ser conectado a tierra y el terminal positivo debería ser conectado a una fuente de alimentación, la cual puede ser cualquiera capaz de aportar un voltaje de entre 2 V y 5 V (si es necesario, a través de

un divisor de tensión). La señal de audio recibida la obtendríamos del terminal positivo, siempre a través de un condensador (de entre 0,1µF y 1µF) actuando como filtro pasa-altos para eliminar cualquier colchón DC de la señal AC obtenida.

Desgraciadamente, conectar un micrófono electret no es tan sencillo porque ya sabemos que es necesario pre-amplificar la señal recibida para hacer que esta sea usable. Por tanto, deberemos construir un circuito accesorio alrededor de él que realice esta función. Un ejemplo muy sencillo es el siguiente, en el cual tan solo hemos utilizamos unas cuantas resistencias y un transistor NPN (como por ejemplo el 2N3904), cuya función es precisamente amplificar la señal, tal como estudiamos en el capítulo anterior dentro del apartado de generación de sonidos mediante *tone()*):



La resistencia R1 ejerce como divisor de tensión del micrófono electret y el condensador C1 sirve, tal como ya hemos comentado, para eliminar el posible colchón DC de la señal AC recibida. A partir de aquí, el funcionamiento lo deberíamos conocer: la señal recibida por el micrófono se envía a la base del transistor, el cual dejará fluir más o menos corriente entre colector y emisor según sea la intensidad de aquella. Si esa corriente generada la tomamos como salida del circuito, tendremos una representación calcada de la señal original pero con mayor amplitud.

La función de las resistencias R2 y R3 es descentrar la señal AC recibida por la base del transistor proveniente del micrófono. Dicho de otra forma: añaden a esa señal un "colchón" DC constante que permite tener a todos los valores de la onda AC

(incluyendo los picos negativos) por encima de 0 V. En caso de ausencia de sonido, la base del transistor recibirá una determinada corriente DC que mantendrá el transistor en un estado de conducción intermedio permanente llamado “punto-Q”; cuando se detecte sonido, las oscilaciones de la señal fluctuarán alrededor de ese estado de conducción intermedio, sin llegar nunca ni al modo de saturación por un lado ni al modo de corte por otro (debido a que la señal se autorregula: si la corriente por el colector incrementa, decrece la que circula por la base, y por tanto automáticamente la corriente por el colector pasa a reducirse). Por otro lado, debido a la aparición de este nuevo colchón DC, una vez obtenida la señal ampliada es necesario eliminarlo en la medida de lo posible mediante el condensador C2.

Los valores de R2 y R3 deberían elegirse con cuidado, porque de ellos depende fundamentalmente la amplitud de la señal amplificada, la cual deberá ser una u otra según lo que nos interese (nivel de línea, tensión de trabajo de Arduino, etc.). De hecho, el uso más habitual de este circuito (y de todos los pre-amplificadores) es adecuar las señales al nivel de línea para que puedan derivarse a circuitos amplificadores de audio propiamente dichos.

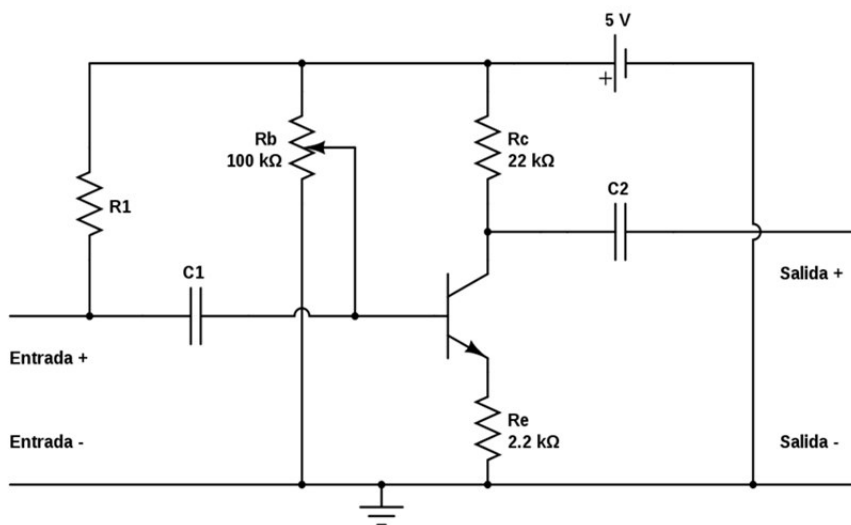
Si conectamos este circuito pre-amplificador a nuestra placa Arduino, y leídos los valores de R2 y R3 mostrados en el diagrama anterior, podremos observar mediante el “Serial monitor” la presencia de un colchón DC de 320 (sobre 1024) y picos de hasta 950 en sonidos muy fuertes. Estos valores ofrecen un rango útil (dentro del rango 0-1023 admitido por las entradas analógicas de la placa Arduino) bastante aceptable, pero si se desea, se pueden probar otros valores de R2, R3 y C2 para mejorarlo (es decir, para reducir más el valor del colchón y aumentar más el rango útil dentro del admitido).

Ejemplo 7.36: Para probar el circuito pre-amplificador anterior podemos ejecutar el código siguiente, el cual nos puede servir, en un entorno silencioso, para observar las fluctuaciones presentes en la señal aun cuando en teoría debiéramos tener una señal constante. Para evitar este fenómeno, podemos utilizar el recurso de calcular la media de los últimos valores leídos (tal como hemos visto en ejemplos anteriores) y así suavizar los picos no deseados.

```
void setup(){
 Serial.begin(9600);
}
void loop(){
 int mini = 1024; //Iré reduciendo el valor de “mini” hasta el real
 int maxi = 0;    //Iré reduciendo el valor de “maxi” hasta el real
```

```
 for(int i=0;i<1000;i++) { //Otra posibilidad: while(millis() < 1000)
       int valor = analogRead(0); //El micro está conectado en pin 0
       mini = min(mini, valor);
       maxi = max(maxi, valor);
 }
 Serial.print("Ruido=");
 Serial.println(maximum-minimum);
}
```

El circuito pre-amplificador presentado no es ni mucho menos el único que existe. Hay literalmente cientos. Otro circuito común (tal vez incluso más que el anterior), es el de la figura siguiente. En él se utiliza un potenciómetro como divisor de tensión para ajustar la corriente deseada a la base del transistor. Lo más práctico es calibrarlo en un entorno silencioso para obtener un voltaje de salida de 2,5 V correspondiente al punto-Q.



Tanto este circuito como el anterior tienen la ventaja de amplificar la señal sin distorsión, pero tiene el inconveniente de que aporta constantemente un colchón DC a la base del transistor, con lo que no es un circuito eficiente energéticamente.

Finalmente, no quisiera dejar de comentar que en muchos foros y blogs de Internet se sugiere utilizar como pre-amplificador un chip bastante popular, el LM386. No es una buena elección, ya que en realidad, el chip LM386 es un amplificador que trabaja a nivel de línea. Es decir, está pensado para aportar, a partir de una entrada

de audio ya pre-amplificada, una potencia de hasta 1 W (en sus versiones más capaces) a un sistema de altavoces. Si se conecta directamente a un micrófono, obtendremos demasiado ruido; una alternativa mejor sería utilizar en todo caso el chip LM358.

Tampoco es buena idea (como a menudo se propone) usar el LM386 como amplificador conectándolo directamente a una salida de audio de nuestra placa Arduino, ya que, como acabamos de decir, este chip está pensado para tener entradas a nivel de línea, las cuales admiten voltajes menores que los 5 V que ofrece un pin-hembra de la placa Arduino.

Reconocimiento de voz

Nuestra placa Arduino es capaz de responder a órdenes expresadas mediante voz gracias al “EasyVR Shield” de Veear (distribuido entre otros por Sparkfun con producto nº 10963), el cual incorpora un módulo de reconocimiento de voz diseñado y fabricado por la misma empresa Veear (también disponible de forma autónoma). Este shield también incluye un micrófono, una salida para conectar un altavoz de 8 ohmios y un zócalo jack de 3,5 mm para conectar unos auriculares.

Este shield se comunica con la placa Arduino a través del canal serie (preferiblemente mediante dos pines RX y TX definidos por software) a una velocidad de 9600 bits/s. Aunque podríamos controlar este shield enviando los comandos adecuados desde un sketch, para gestionar su funcionalidad y comportamiento es mucho más sencillo utilizar la librería que el propio fabricante ofrece en http://www.veear.eu/downloads. La documentación necesaria para aprender su uso viene incluida dentro del paquete descargado.

Este shield incluye de fábrica un conjunto de órdenes predefinidas disponibles en varios idiomas (entre ellos el español), ideales para realizar controles básicos, pero también admite la definición de hasta 32 comandos propios totalmente personalizados, incluyendo contraseñas. Para definir estos comandos y/o configurar las órdenes pregrabadas es necesario utilizar el software gráfico (llamado “EasyVR Commander”) que el propio fabricante ofrece (aunque tan solo para sistema Windows) en http://www.veear.eu/downloads. La información necesaria para aprender su uso la podemos consultar en la Guía de Usuario, disponible en el mismo sitio de descargas.

8 COMUNICACIÓN EN RED

En este capítulo veremos diferentes formas de comunicar nuestra placa Arduino con otras placas (o computadores) conectados a redes de diferentes tipos: concretamente a redes Ethernet cableadas, redes Wi-Fi y redes Bluetooth. El objetivo es controlar y transferir información entre estos dispositivos de forma remota. Así, podríamos, por ejemplo, acceder a datos de sensores o controlar una instalación de actuadores sin tener que desplazarnos físicamente. Para ello supondremos, mientras no se diga lo contrario, que utilizaremos o bien la placa Arduino Ethernet o bien la placa Arduino UNO con el shield Arduino Ethernet acoplado.

CONCEPTOS BÁSICOS SOBRE REDES

Dirección IP

Un dato que siempre ha de tener asignado una placa/shield Arduino Ethernet para que esta tenga conectividad a la red es una dirección IP. De hecho, cualquier dispositivo (como un computador) ha de tener configurada correctamente una dirección IP propia para poder formar parte de una red TCP/IP.

La dirección IP es una etiqueta numérica formada por cuatro cifras, de valores entre 0 y 255 separados por un punto, que identifica a la tarjeta de red de un dispositivo (computador, placa Arduino Ethernet, etc.) dentro de la red de tipo TCP/IP. Cada tarjeta tiene una dirección IP exclusiva, por lo que, utilizando estas direcciones

los dispositivos pueden reconocerse y comunicarse entre sí. Un ejemplo de ip podría ser 192.168.0.1.

En un computador, la dirección IP se puede establecer manualmente por el usuario (lo que se llama usar una “ip fija” o “ip estática”) mediante diferentes utilidades específicas, distintas según el sistema operativo utilizado, o bien puede existir en la red un dispositivo especializado en conceder automáticamente direcciones ip al resto de dispositivos cuando estos la soliciten (lo que se llama usar una “ip dinámica”). Esta solicitud puede realizarse por decisión del usuario o bien, más frecuentemente, de forma automática durante el arranque del computador mediante un protocolo de intercambio de mensajes llamado DHCP. Cuando se utiliza este protocolo de solicitud-concesión de ips, al dispositivo que concede la ip se le suele llamar “servidor DHCP” y al dispositivo que la solicita se le llama “cliente DHCP”. Ambos tipos de ip (fija o dinámica), independientemente de cómo se hayan establecido, funcionalmente son idénticas.

La placa Arduino Ethernet (y similares) puede adquirir su ip también de estas dos maneras: bien de forma fija estableciendo su valor concreto dentro del propio código de nuestro sketch, bien de forma dinámica obteniéndola de algún servidor DHCP existente en la red. En todo caso, hemos de usar la librería oficial “Ethernet”.

Se sale de los objetivos de este libro el detallar cómo se configura de forma fija la ip en computadores ejecutando diferentes sistemas operativos. Remito a la ayuda oficial de cada sistema. Igualmente, tampoco detallaremos cómo se puede instalar y administrar un servidor DHCP. Solo como referencia, comentaremos la existencia de algunas aplicaciones que permitan convertir un computador en un servidor DHCP: en Linux podemos usar el software “ISC Dhcpd” (http://www.isc.org/software/dhcp) o el “Dnsmasq” (http://www.thekelleys.org.uk); en Windows podemos usar el que viene incorporado de fábrica en sus versiones Server o bien el DhcpServer (http://www.dhcpserver.de/dhcpsrv.htm) o el “Open Dhcp Server” (http://sourceforge.net/projects/dhcpserver) o también el “Dual Dhcp-Dns Server” (http://dhcp-dns-server.sourceforge.net), entre otros. Remito a su respectiva documentación oficial para conocer cómo ponerlos en marcha.

Máscara de red

La máscara de red sirve para identificar a qué red pertenece un dispositivo que tenga una dirección ip concreta. Un dispositivo (por ejemplo, nuestra placa Arduino) solamente puede pertenecer en un momento determinado a una única red. Saber a qué red pertenece un dispositivo es muy importante, porque solamente dispositivos de la misma red son capaces de comunicarse entre sí.

Una máscara de red es un conjunto de cuatro cifras de valores entre 0 y 255 separados por un punto. Existen muchos tipos de máscara de red, pero nosotros nos centraremos en las tres más básicas: la máscara de clase A (cuyo valor es 255.0.0.0), la de clase B (cuyo valor es 255.255.0.0) y la de clase C (cuyo valor es 255.255.255.0).

Para saber a qué red concreta pertenece un dispositivo con una determinada ip y máscara, debemos conocer su identificador de la red, el cual es también un conjunto de cuatro cifras entre 0 y 255 separadas por un punto. Este identificador se forma eligiendo la parte de la ip del dispositivo que coincide con la parte de su máscara de valor 255, y luego añadiendo 0 a la parte que coincide con la parte de su máscara de valor 0. Por ejemplo, si tenemos una placa Arduino (o un computador, es lo mismo) que tiene la ip 192.168.23.1 y una máscara de 255.255.0.0, la red a la que pertenece será "192.168.0.0", y se podrá comunicar con todos los dispositivos que pertenezcan a esa misma red (como por ejemplo, suponiendo siempre la misma máscara, el que tenga una ip como 192.168.142.62 o 192.168.216.39, etc.). En cambio, ese dispositivo no se podrá comunicar con otro que por ejemplo tenga la ip 192.76.23.123 (y la misma máscara), porque este pertenecería a la red "192.76.0.0", que es diferente.

En un computador, la máscara de red se puede establecer manualmente por el usuario mediante las mismas aplicaciones que permitían establecer el valor de una "ip fija" (las cuales ya hemos comentado que según el sistema operativo utilizado son diferentes), o bien puede ser asignada mediante el servidor DHCP existente en la LAN. En una placa/shield Arduino Ethernet, la máscara se puede establecer escribiéndola "a mano" dentro del código de nuestro sketch (mediante la librería "Ethernet"), o bien ser asignada a través de algún servidor DHCP existente en la red.

Direcciones IP privadas

A la hora de asignar una ip a nuestra placa, nos puede venir la duda de si cualquier combinación de cuatro números es válida. La respuesta es no. Para empezar, cada uno de los cuatro números solamente puede tener un valor entre 0 y 255, pero ni siquiera son válidas todas las combinaciones posibles de esos valores: solamente podemos utilizar una ip que sea de tipo "privada". Expliquemos esto.

Cada dispositivo conectado directamente a Internet tiene una ip "pública" diferente que le permite comunicarse con el resto de dispositivos del mundo. El IANA (http://www.iana.org) es el organismo internacional que asigna de forma controlada estas ips públicas a las entidades que necesiten disponer de acceso directo a Internet (como los operadores telefónicos, entre otros). Pero el IANA solo trabaja con

organizaciones reconocidas internacionalmente: nunca concede ips públicas a usuarios finales. Así que nosotros como usuarios no podremos nunca administrar ips públicas. De hecho, el acceso doméstico a Internet es posible gracias a que el operador que tenemos contratado nos ofrece una de las ips públicas que él ha obtenido de parte de la IANA.

No obstante, suele ser bastante habitual que en una empresa u organización (o incluso en un domicilio particular) tengamos varios equipos y todos queramos conectarlos entre sí y a Internet. Además de que sería un gran desperdicio asignar a cada uno de estos equipos una ip pública (mas teniendo en cuenta que estas no son infinitas y que actualmente se están agotando), acabamos de decir que esto es imposible porque la IANA no nos lo permite. La solución es utilizar ips "privadas". Es decir, ips que solo puedan funcionar en el interior de una red local, sin poder "salir afuera" (es decir, sin poder acceder directamente a Internet).

En realidad, el acceso a Internet sí que es posible, aunque todos los equipos de nuestra LAN usen una ip privada, gracias a la existencia de un equipo intermediario (lo que comúnmente llamamos "router") que es el único que tiene asignada una ip pública y que es utilizado por el resto de los equipos de la LAN como pasarela al exterior. Con este "truco" conseguimos que múltiples máquinas tengan acceso a Internet usando una sola ip pública, camuflando todo el interior de la LAN.

Lo bueno de este sistema es que, además de ahorrar ips públicas, las ips privadas se pueden reutilizar todas las veces que se desee en diferentes LANs, porque estas últimas no se ven entre sí: se quedan confinadas en el interior de la LAN. Esto hace que podamos asignar a los equipos de nuestra organización unas ips privadas y que otra persona de cualquier otra parte del mundo asigne las mismas en su propia organización, sin ningún problema: no habrá ningún conflicto porque entre ambas organizaciones tan solo se ven sus ips públicas respectivas (concedidas por sus respectivos operadores) y por tanto no hay ningún tipo de solapamiento.

Así pues, resumiendo: las únicas ips que podemos asignar a nuestra placa Arduino para comunicarla con las máquinas de nuestra red local son ips privadas. Existen oficialmente varias redes reservadas para un uso exclusivamente privado. La red concreta que elijamos da igual, pero en cualquier caso todos los dispositivos de nuestra LAN han de pertenecer a la misma red para que se puedan "ver" entre sí. Las redes privadas posibles a elegir son:

La **red "10.0.0.0"** de clase A (255.0.0.0): Es decir, en nuestros dispositivos podemos usar ips que vayan desde la 10.0.0.1 hasta la 10.255.255.254 (la

10.255.255.255 es una ip especial y en nuestros proyectos no la utilizaremos). Todos estos equipos pertenecen a la misma red (la 10), y se puede comprobar fácilmente contando las ips posibles que en ella pueden existir miles de equipos.

Las **redes "172.16.0.0", "172.17.0.0", "172.18.0.0"...hasta la "172.31.0.0"** de clase B (255.255.0.0) : En cada una de estas redes podemos usar ips que vayan desde x.x.0.1 hasta x.x.255.254 (es decir, desde 172.16.0.1 hasta 172.16.255.254, ó desde 172.17.0.1 hasta 172.17.255.254, etc). Se puede ver fácilmente que podemos utilizar hasta 16 redes privadas diferentes de clase B (a diferencia de la única red privada de clase A posible), pero en cada una de estas redes de clase B pueden existir menos cantidad de equipos.

Las **redes "192.168.0.0", "192.168.1.0", "192.168.2.0"... hasta la "192.168.255.0"** de clase C (255.255.255.0). En cada una de estas redes podemos usar ips que vayan desde x.x.x.1 hasta x.x.x.254 (es decir, desde 192.168.0.1 hasta 192.168.0.254, ó desde 192.168.1.1 hasta 192.168.1.254, etc). Se puede comprobar que podemos utilizar hasta 256 redes privadas diferentes de clase C, pero en cada una de ellas tan solo pueden existir hasta 254 equipos.

Dirección MAC

Otro dato imprescindible para que la placa Arduino Ethernet se pueda conectar a la red, además de una dirección IP y su máscara de red, es que tenga una dirección MAC. De hecho, cualquier computador ha de tener siempre especificada una dirección MAC propia para poder formar parte de una red TCP/IP.

La dirección MAC es una etiqueta de 48 bits (12 caracteres hexadecimales) que identifica a la tarjeta de red de manera única e inequívoca en el mundo. Este dato no depende del protocolo de conexión utilizado ni de la red: es un valor fijado por el fabricante de la tarjeta que (normalmente) no se puede cambiar porque viene grabado en el hardware de esta. Afortunadamente, en la mayoría de los dispositivos (como es el caso de los computadores) no es necesario conocer (y ni mucho menos cambiar) la dirección MAC ni para montar una red doméstica ni para configurar la conexión a Internet ni nada, porque esta solo se usa a niveles más internos de la red y viene predefinida de fábrica. Un ejemplo de dirección MAC podría ser 12-AB-56-78-90-FE.

En el caso de la placa/shield Arduino Ethernet, no obstante, sí que debemos especificar en el código de nuestro sketch su dirección MAC (mediante la librería

"Ethernet"). Dependiendo de la antigüedad del modelo de placa/shield que tengamos, puede ser que tenga o no la dirección MAC predefinida de fábrica. En el caso de que sea así, esta MAC se mostrará impresa en una etiqueta pegada a la placa/shield y en el código de nuestro programa deberemos utilizar dicha dirección MAC. Si no vemos ninguna etiqueta, la placa/shield no tendrá ninguna MAC predefinida, por lo que en el código de nuestro sketch nos la deberemos inventar (procurando que no coincida con ninguna otra que tengan los dispositivos conectados a nuestra red local en ese momento).

Servidores DNS

Un servidor DNS es un computador (normalmente de acceso público a través de Internet) que hace posible que los usuarios utilicen nombres descriptivos en lugar de direcciones ip (más difíciles de aprender y recordar) para identificar y conectar con los distintos equipos presentes en la red. Es decir, son computadores ubicados en diferentes partes del mundo que permiten que los usuarios de Internet puedan usar un nombre sencillo (como por ejemplo www.rclibros.es) para conectar con un ordenador concreto en lugar de escribir su dirección ip (como 82.98.148.182).

Cuando un usuario escribe un nombre DNS en una aplicación de nuestro computador (como un navegador), lo primero que ocurre es que esa aplicación consulta qué servidor DNS (o servidores, ya que puede haber varios) tiene predefinido el sistema operativo utilizado. Cuando descubre la ip guardada de ese servidor DNS predefinido, le envía una consulta solicitando conocer cuál es la dirección ip real que se corresponde con el nombre escrito por el usuario. Si el servidor DNS le responde con la información solicitada, la aplicación puede entonces comunicarse directamente con ese equipo remoto usando su dirección ip. Si el servidor DNS no tiene ninguna entrada en su base de datos para el nombre consultado, normalmente él mismo consultará a otro servidor DNS hasta que se encuentre uno que sí conozca la correspondencia nombre <-> ip, o bien hasta que se descubra que el nombre consultado no pertenece a ninguna máquina existente.

En un computador, los servidores DNS a consultar de forma predeterminada se pueden establecer manualmente por el usuario mediante las mismas aplicaciones que permitían establecer el valor de una ip fija o la máscara (las cuales ya hemos comentado que según el sistema operativo utilizado son diferentes), o también pueden ser asignados mediante el servidor DHCP existente en la LAN. En el caso de la placa/shield Arduino Ethernet, el servidor DNS que queramos utilizar se puede establecer escribiendo su ip "a mano" dentro del código de nuestro sketch (mediante la librería "Ethernet"), o bien puede ser asignado a través de algún servidor DHCP existente en la red.

Hay que aclarar que si un dispositivo (placa/shield Arduino Ethernet o computador) no tiene configurado ningún servidor DNS, podrá seguir comunicándose con el resto de equipos utilizando sus direcciones ip directamente. Es decir, el uso de servidores DNS no es imprescindible técnicamente hablando, aunque no hay duda que facilita mucho el uso de la red por parte de los usuarios.

Algunos de los servidores DNS públicos que podemos utilizar en nuestros sketches de Arduino pueden ser los de Google (con ips 8.8.8.8 y 8.8.4.4), los de OpenDNS (208.67.222.222 y 208.67.220.220), los de DNSAdvantage (156.154.70.1 y 156.154.71.1) o los de ScrubIT (67.138.54.100 y 207.225.209.66), entre otros muchos (como los proporcionados por cada operador telefónico).

Es posible instalar en nuestra propia LAN un software específico para convertir un computador en servidor DNS, como la aplicación “ISC Bind” (http://www.isc.org/software/bind) o el “Dnsmasq” (http://www.thekelleys.org.uk), pero esto no lo tendremos que hacer a no ser que necesitemos que los equipos de nuestra red local también tengan nombres propios. Normalmente, los servidores DNS que se suelen utilizar son públicos de Internet (como los listados en el párrafo anterior), para poder usar así los nombres de todos los ordenadores públicos de Internet (servidores web, servidores de correo, etc.), que es lo que generalmente querremos.

Puerta de enlace predeterminada

Una puerta de enlace predeterminada (también llamado “gateway”) es un dispositivo especializado en comunicar dos o más redes entre sí (es decir, en conectarlas y redirigir el tráfico de datos entre ellas). Generalmente, en las casas u oficinas, este dispositivo (al que comúnmente se le llama “router” o enrutador) conecta la red local del domicilio con Internet. En las empresas, muchas veces esta función recae en un computador que, además de redirigir el tráfico de datos entre la red local y la red exterior (Internet), realiza más tareas (como hacer de cortafuegos, por ejemplo). En cualquier caso, un “router” estándar debe incorporar internamente una tarjeta de red con una ip pública asignada por el operador telefónico (que servirá para identificarse dentro de Internet), y otra tarjeta de red con una ip privada (que servirá para identificarse dentro de la red local para ser accesible así al resto de equipos de esa red).

Todos los equipos de una red local deberán tener configurada la ip privada de la puerta de enlace predeterminada para que puedan saber a dónde dirigir los mensajes destinados al exterior. En el caso de computadores ejecutando diferentes

sistemas operativos, se sale de los objetivos de este libro el detallar cómo se realiza esto: remito a la ayuda oficial de cada sistema. En el caso de la placa/shield Arduino Ethernet, la puerta de enlace predeterminada que queramos utilizar se puede establecer escribiendo su ip "a mano" dentro del código de nuestro sketch (mediante la librería "Ethernet"), o bien puede ser asignado a través de algún servidor DHCP existente en la red.

Hay que aclarar que si un equipo no tiene configurado una puerta de enlace, podrá seguir comunicándose con el resto de equipos de su propia red local, pero en el momento que necesite enviar un mensaje al exterior (a otra red), no sabrá a quién pasárselo y por tanto no se podrá comunicar con otras redes.

USO DE LA PLACA/SHIELD ARDUINO ETHERNET

Configuración inicial de los parámetros de red

Para que una placa Arduino Ethernet (o el shield Ethernet o alguna otra placa/shield que incorpore el mismo chip Wiznet W5100) pueda empezar a utilizar una red TCP/IP, lo primero que se ha de hacer es asignarle una serie de valores de configuración (dirección MAC, dirección ip, etc.). Para ello escribiremos dentro del "setup()" de nuestro código la función *Ethernet.begin()*. Esta función tiene varias formas de escribirse:

Ethernet.begin(mac): donde "mac" representa la dirección MAC que ha de tener nuestra tarjeta. En los modelos más modernos del shield Arduino Ethernet, esta dirección viene impresa en una etiqueta, y por tanto, se ha de poner ese valor, pero en shields más antiguos se puede elegir una cualquiera. Cualquier valor de una dirección MAC es un array de tipo "byte" de seis elementos escritos en formato hexadecimal, por lo que normalmente declararemos e inicializaremos previamente ese array en la zona de declaraciones globales (por ejemplo, así: `byte mimac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };`) y luego asignaremos dentro de la función "setup()" esa dirección MAC a nuestra placa/shield escribiendo algo como esto simplemente: `Ethernet.begin(mimac);`.

Esta forma de escribir la función *Ethernet.begin()* –es decir, tan solo especificando la dirección MAC y ya está– provoca que la placa/shield solicite vía DHCP una ip a algún servidor existente en la red local. Por tanto, se ha de haber configurado un computador dentro de nuestra LAN para que actúe como servidor

DHCP. Si la conexión con el servidor DHCP se establece correctamente, la función *Ethernet.begin()* retornará un valor de tipo "int" con valor 1, y si no, retornará 0. El resto de formas de *Ethernet.begin()* explicadas a continuación no tienen valor de retorno.

Por otro lado, esta forma de escribir la función *Ethernet.begin()* especificando tan solo la dirección MAC incrementa notablemente el tamaño del sketch final compilado, asunto que debemos tener en cuenta para no sobrepasar el límite impuesto por la memoria del microcontrolador.

Ethernet.begin(mac, ip): donde "ip" representa la dirección ip (fija, en este caso) que asignamos manualmente a la placa/shield Arduino Ethernet. Cualquier valor de una dirección ip es un array de tipo "byte" de cuatro elementos, escritos en formato decimal, por lo que normalmente declararemos e inicializaremos previamente ese array en la zona de declaraciones globales (por ejemplo, así: `byte miip[] = { 10, 0, 0, 17 };`) y luego asignaremos dentro de la función "setup()" esa dirección ip a nuestra placa/shield escribiendo algo como esto simplemente: `Ethernet.begin(mimac,miip);` –suponiendo que "mimac" ha sido declarado como el array que contiene la dirección MAC–.

Existe otra manera de declarar direcciones ip en vez de utilizar un array de tipo byte: mediante la declaración especial `IPAddress miip(10,0,0,17);` (donde en este ejemplo hemos llamamos a nuestra ip "miip" y le hemos asignado el valor 10.0.0.17).

Ethernet.begin(mac, ip, servdns): esta forma de escribir la función es muy similar a la anterior. Simplemente se le añade a la placa/shield Arduino Ethernet un dato de configuración suplementario (además de su dirección MAC y su dirección ip fija) que es la dirección ip de un servidor DNS que esta placa utilizará. Esta dirección ip es un array de tipo "byte" de cuatro elementos, escritos en formato decimal, declarado e inicializado previamente en la zona de declaraciones globales.

Ethernet.begin(mac, ip, servdns, gateway): esta forma de escribir la función es muy similar a la anterior. Simplemente se le añade a la placa/shield Arduino Ethernet un dato de configuración suplementario (además de su dirección MAC, su dirección ip fija y la dirección de un servidor DNS predefinido), que es la dirección ip de la puerta de enlace ("gateway") de nuestra red local que esta placa utilizará. Esta dirección ip es un array de tipo

"byte" de cuatro elementos, escritos en formato decimal, declarado e inicializado previamente en la zona de declaraciones globales.

En las formas de *Ethernet.begin()* donde no se especificaba explícitamente la dirección ip de la puerta de enlace (las tres primeras), esta se asigna por defecto a la misma ip de la propia placa Arduino excepto en el último número de los cuatro, que se establece a 1. Es decir, si la ip de la placa decidimos que sea 10.0.0.17, la ip del gateway se establece automáticamente, si no se indica lo contrario, a 10.0.0.1.

Ethernet.begin(mac, ip, servdns, gateway, subnet): esta forma de escribir la función añade a las formas anteriores la posibilidad de configurar además la máscara de red que tendrá la placa/shield Arduino Ethernet. Este dato es un array de tipo "byte" de cuatro elementos, escritos en formato decimal, declarado e inicializado previamente en la zona de declaraciones globales. Su valor por defecto, si no se especifica explícitamente, es 255.255.255.0

En el caso de que hayamos utilizado la primera forma de *Ethernet.begin(),* es decir, el que utiliza un servidor DHCP externo para obtener la ip de la placa/shield, nos será interesante conocer dos funciones más:

Ethernet.localIP(): devuelve la dirección ip de la placa/shield. Este valor de retorno ha de ser declarado previamente de tipo "IPAddress". Si la ip es fija no tendrá mucho sentido utilizar esta función, pero en los casos donde esta ip es dinámica, es la manera de saber exactamente qué ip ha obtenido nuestra placa/shield en un momento determinado. No tiene parámetros.

Ethernet.maintain(): permite la renovación de la concesión de una ip dinámica. Cuando una ip es asignada a un dispositivo por un servidor DHCP, esta asignación es válida durante un determinado tiempo decidido por el servidor. Con esta función es posible solicitar una renovación en la asignación de la ip concedida. Dependiendo de la configuración del servidor, se renovará efectivamente esa misma ip, o bien se asignará una nueva ip diferente, o bien no se concederá la prórroga. Esta función no tiene parámetros. Su valor de retorno es de tipo "byte" y puede ser: 0 (si no pasa nada), 1 (si la concesión de una nueva ip ha fallado), 2 (si la concesión de una nueva ip se ha realizado con éxito), 3 (si la renovación de la ip existente ha fallado), 4 (si la renovación de la ip existente se ha realizado con éxito).

La librería Ethernet también ofrece la posibilidad de establecer conexiones de tipo UDP (mediante objetos de tipo "EthernetUDP"), pero en este libro no las

estudiaremos. Si se desea aprender su uso, remito a la documentación oficial disponible en la página web de Arduino.

Uso de Arduino como servidor

Las redes TCP/IP suelen tener una arquitectura llamada de "cliente-servidor". Esto significa que en la comunicación establecida entre dos máquinas, una de ellas toma el rol de "cliente" y la otra de "servidor". La diferencia está en su comportamiento: lo que hace un cliente es realizar peticiones puntuales a un servidor, y lo que hace un servidor es recibir estas peticiones y ofrecer una respuesta adecuada al cliente como respuesta. El servidor, por tanto, ha de estar permanentemente "escuchando" la posibilidad de recibir las peticiones de los clientes, que se pueden producir en cualquier momento y pueden provenir de múltiples clientes a la vez.

Las peticiones de los clientes pueden ser de muchos tipos: por ejemplo, un cliente puede solicitar la descarga de un fichero almacenado en el servidor, otro cliente puede solicitar imprimir con una impresora conectada al servidor, otro cliente puede solicitar ver la página web que esté alojada en el servidor, etc. En general, pues, un servidor de un tipo concreto ofrece un recurso concreto para clientes de ese mismo tipo. Así, tendremos servidores y clientes de ficheros, servidores y clientes de impresión, servidores y clientes web, etc.

Un mismo computador puede ejercer de servidor de varios tipos (web, de impresión, de ficheros, etc.). Para evitar que las peticiones de clientes de distintos tipos interfieran entre sí cuando conecten con un servidor "multitipo", existe un mecanismo que es el siguiente: cada recurso (servicio) ofrecido por el servidor viene "identificado" por un número (el llamado "número de puerto"). A través de un puerto concreto, el servidor solamente aceptará peticiones de un tipo concreto de clientes, de forma que en un servidor puede haber varios puertos abiertos, y cada uno de ellos escuchando posibles solicitudes de un tipo de clientes, sin mezclarse. Los puertos son algo parecido a las ventanillas de un banco: si suponemos que los clientes del banco representan las peticiones, cada una de ellas ha de ir dirigida a la ventanilla que toque según el tipo de petición que sea (si es para sacar dinero se tendrá que ir a una ventanilla, si es para contratar una hipoteca, a otra, y así); de esta forma, las peticiones de distinto tipo no interfieren entre sí y todo funciona mucho más ordenada y eficazmente. Es muy importante que cada cliente se conecte al puerto que le toca, porque si no, es posible que la comunicación sea imposible entre él y el servidor, ya que este puede no reconocer la petición que le esté llegando.

Cuando decimos que la placa Arduino actúe como servidor, estaremos ofreciendo algún tipo de recurso de forma permanente para que cualquier dispositivo con capacidad de conectarse a ella (otra placa, un computador, etc.) pueda disponer de él. Esto implica que nuestra placa deberá tener abierto un puerto –como mínimo– para poder escuchar y contestar esas solicitudes recibidas.

Lo primero que debemos hacer para convertir nuestra placa Arduino en un servidor es declarar, en la zona de declaraciones globales, una variable (en realidad, crear un objeto) de tipo "EthernetServer". Esto se hace usando la siguiente sintaxis (suponiendo que llamamos "miservidor" a dicha variable): *EthernetServer miservidor(nº puerto);* donde "nº puerto" es un número entero que representa el número de puerto usado para escuchar las peticiones de los clientes que se conecten al servidor.

Una vez ya creado el objeto "miservidor" en la declaración anterior, lo deberemos poner en marcha mediante la siguiente función:

> **miservidor.begin()**: hace que "miservidor" empiece a escuchar las peticiones recibidas a través del puerto que hayamos definido en su declaración. Normalmente, esta función se ejecuta dentro de "setup()", justo después de *Ethernet.begin()*. No tiene ni parámetros ni valor de retorno.

A partir de aquí, ya podemos hacer cosas en nuestro sketch. Por ejemplo, podemos enviar datos a todos los clientes (sin distinción) que estén ya conectados a nuestra placa. Para ello, disponemos de tres funciones diferentes:

> **miservidor.print()**: envía el dato especificado como parámetro a todos los clientes conectados en este momento a la placa-servidor. Este dato puede ser de cualquier tipo: entero, decimal, carácter o cadena de caracteres. Si es numérico, será tratado como una secuencia de caracteres ASCII (es decir, el número 123 se enviará como tres caracteres: '1', '2' y '3'). Opcionalmente, tiene un segundo parámetro, útil en el caso de que el dato a enviar sea entero, el cual puede valer alguna de las siguientes constantes predefinidas: BIN (para enviar el dato en formato binario), HEX (para enviarlo en formato hexadecimal) o DEC (para enviarlo en formato decimal, aunque por defecto ya es así). Su valor de retorno es el número de bytes enviados, pero utilizarlo es opcional.

> **miservidor.println()**: funciona exactamente igual a *miservidor.print()*, con la diferencia de que al final del envío del dato pasado como parámetro, añade los caracteres ASCII 10 y 13, provocando un salto de línea.

miservidor.write(): envía el dato especificado como parámetro a todos los clientes conectados en este momento a la placa-servidor. Este dato, sin embargo, solo puede ser de tipo "char" o "byte". No tiene valor de retorno.

Pero lo más interesante es tratar individualmente con cada una de las conexiones que se produzcan de una forma independiente y "personalizada". (recordemos que un puerto de una placa/shield Ethernet actuando como servidor puede soportar hasta cuatro clientes conectados simultáneamente). Para ello, necesitaremos la función:

miservidor.available(): devuelve (crea) un objeto de tipo "EthernetClient" cuando "miservidor" detecta una entrada de datos proveniente de algún cliente exterior. Este objeto representa la conexión establecida con ese cliente. Si "miservidor" no recibe ninguna entrada de datos, esta función devuelve 0 y por tanto no se crea ningún objeto. En cualquier caso, este objeto "EthernetClient" ha de ser previamente declarado (en el ámbito donde se desee de nuestro sketch) mediante la sintaxis: `EthernetClient micliente;` (suponiendo que llamamos "micliente" a dicho objeto). Una vez que *miservidor.available()* haya creado el objeto "micliente", la conexión que este representa podrá ser manipulada mediante una serie de instrucciones propias de "micliente" y que solo afectan a esa conexión. De esta manera podremos enviar o recibir datos comunicándonos exclusivamente con "micliente". Importante hacer notar que la conexión gestionada por "micliente" es persistente, y para cerrarla explícitamente es necesario utilizar la función *micliente.stop().* Esta función no tiene parámetros.

Una vez creado el objeto "micliente", podremos comunicarnos con ese cliente particular de diversas formas. En concreto, para recibir datos de él (de hecho, los que se han detectado mediante *miservidor.available()*) usaremos:

micliente.read(): devuelve un byte proveniente de "micliente". Cada vez que se ejecute esta función, devolverá el siguiente byte recibido de esa conexión. Si ya no hay más bytes disponibles para leer, devolverá -1. Esta función no tiene parámetros.

micliente.flush(): elimina todos los bytes que han llegado al buffer de entrada del servidor provenientes de "micliente" y que aún no habían sido leídos. No tiene ni parámetros ni valor de retorno.

Y para enviar datos desde "miservidor" a "micliente" tenemos varias posibilidades:

micliente.print(): funciona exactamente igual que *miservidor.print()* pero esta vez solo para la conexión “micliente”.

micliente.println(): funciona exactamente igual que *miservidor.println()* pero esta vez solo para la conexión “micliente”.

micliente.write(): funciona exactamente igual que *miservidor.write()* pero esta vez solo para la conexión “micliente”.

Ejemplo 8.1: Con un ejemplo de código se ve todo más claro. Para probar el siguiente sketch necesitaremos una placa/shield Arduino Ethernet conectada a una red local (normalmente, mediante un switch de red) y uno o más computadores conectados también a la misma red (los cuales harán de cliente). El código siguiente reenvía los datos que recibe por el puerto abierto (que es el 23, pero podría haber sido otro cualquiera) de un cliente particular hacia todos los clientes (incluyendo también el remitente) que estén conectados en ese momento. Ese reenvío también lo realiza a través del canal serie hacia el “Serial monitor” (o equivalente), por lo que si queremos ver esos datos recibidos deberemos conectar con un cable USB la placa/shield Arduino Ethernet a nuestro computador.

```
/*Se necesita incluir la librería SPI porque el módulo Wiznet se comunica con la placa a través de este protocolo de comunicación. Nuestro sketch no hace uso explícitamente de las instrucciones de esta librería, pero las instrucciones de la librería Ethernet internamente sí. */
#include <SPI.h>
#include <Ethernet.h>
//La dirección MAC de nuestra placa/shield
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
//La dirección ip de nuestra placa/shield
byte ip[] = { 192, 168, 1, 177 };
/*El puerto 23 es usado en servidores Telnet,
pero puede ser otro*/
EthernetServer miservidor(23);
/*Declaro el objeto “micliente” para poder
gestionarlo en el sketch, cuando se conecte*/
EthernetClient micliente;
void setup() {
       //Se inicializa la placa/shield
       Ethernet.begin(mac, ip);
       //La placa empieza a escuchar por el puerto 23
       miservidor.begin();
```



```
      Serial.begin(9600);
}
void loop() {
      int dato;
/*La primera línea mira se ha recibido bytes por el puerto 23. Si es
así ,se crea el objeto "micliente" correspondiente a esa conexión y
se pasa a ejecutar el interior del "if". Si no es así, se vuelve al
principio del loop() otra vez y continuar mirando si se reciben bytes
por el puerto 23, de forma infinita.*/
      micliente = miservidor.available();
      if (micliente > 0) {
/*Se lee byte a byte lo que llega del cliente "micliente" para
reenviarlos a todos los clientes conectados a "miservidor" en ese
momento y al canal serie de la placa/shield*/
            dato=micliente.read();
            miservidor.write(dato);
            Serial.write(dato);
      }
}
```

Para probar el sketch anterior necesitamos que un computador-cliente se conecte al puerto 23 de la placa/shield Arduino y envíe algún dato. Existen muchos programas que pueden hacer esto, pero los más sencillos son los clientes Telnet de consola, que suelen venir incluidos "de fábrica" en la mayoría de sistemas operativos. Para usarlos, ya sea en Windows o en Linux, debemos abrir el terminal de comandos y escribir *telnet ipplaca nº puerto*. Es decir, en nuestro caso, `telnet 192.168.1.177 23`. A partir de allí, cualquier dato escrito, cuando se pulse "enter" será enviado al servidor Telnet (es decir, la placa Arduino). Es importante tener en cuenta además que para que el computador-cliente pueda hacer la conexión correctamente dentro de la red local, ha de tener una ip de la misma red que la placa Arduino y la misma máscara. Es decir, que debería tener una ip del tipo 192.168.1.x, donde "x" es un número entre 1 y 254 (que no sea el 177, que es el que tiene la placa Arduino) y debería tener una máscara tal como 255.255.255.0.

Además de los clientes Telnet de consola, se podrían utilizar otros programas. Por ejemplo, en Windows podemos utilizar un cliente Telnet (y mucho más) con configuración gráfica llamado Putty (http://www.putty.org). En Linux podemos utilizar el comando NetCat (http://netcat.sourceforge.net), herramienta muy versátil y flexible que puede actuar como un sustituto más capaz de Telnet.

El uso de ips públicas para acceder a Arduino

El ejercicio anterior solo funciona si nuestra placa/shield Arduino y los computadores-cliente pertenecen a la misma red LAN. Pero ¿no es posible conectar nuestra placa/shield a Internet, de forma que pudiera recibir mensajes desde cualquier computador del mundo? Para ello, nuestra placa/shield debería tener una ip pública. Podemos conseguir el mismo efecto “vinculando” la ip pública que tiene la puerta de enlace de nuestra red (nuestro “router”) a la placa/shield Arduino para que cuando el router reciba un mensaje se lo reenvíe a ella. Esto es un procedimiento que se ha de realizar entrando en el panel de control del router (normalmente vía web), y varía de modelo a modelo. No obstante, no lo estudiaremos porque tiene un inconveniente: la gran mayoría de las veces, las ips públicas asignadas por los operadores telefónicos van cambiando de vez en cuando según sus criterios, por lo que la ip pública de nuestro router un día puede ser una y otro día puede ser otra. Esto quiere decir que los computadores-cliente deben saber en cada momento qué ip pública se corresponde con nuestra placa/shield Arduino. Y esto no es demasiado práctico.

Una manera sencilla de evitar esta complicación es utilizar un servicio de DNS dinámico (DDNS), como el ofrecido por http://www.no-ip.com o similar. La idea es vincular la ip pública de nuestro router con un nombre DNS elegido por nosotros, de manera que los computadores-cliente no tengan que conocer nuestra ip pública para conectarse, sino que baste con utilizar ese nombre DNS. Lo importante es saber que este servicio automáticamente actualizará la vinculación ip pública<->nombre DNS cada vez que nuestra ip pública cambie, por lo que no nos tendremos que preocupar nunca de este problema. Así pues, usando un servidor DNS dinámico, nuestra puerta de enlace tendrá un nombre accesible para todos los computadores del mundo.

La manera de utilizar concretamente el servicio No-Ip es muy sencilla. Primero deberemos crear una cuenta de usuario y seguidamente deberemos loguearnos con ella en su web. Una vez allí dispondremos de la opción “Add Host”, la cual nos llevará a un formulario donde podremos elegir el nombre DNS que queremos para nuestro router. A ese nombre se le añadirá siempre una “coletilla” propia de No-Ip, que también podemos elegir de entre varias que hay en un cuadro desplegable. Por ejemplo, un nombre podría ser “mirouter.no-ip.org”, o “mirouter.zapto.org”, dependiendo de la coletilla seleccionada. En ese mismo formulario hay que asegurarse de que esté marcada la opción “DNS Host (A)” , que ya lo estará por defecto, y nada más. La ip que se nos muestra en ese formulario se corresponde a la ip pública detectada de nuestro router; la podemos especificar a mano si la conocemos y sabemos que la mostrada es incorrecta, pero esto es poco

probable. Una vez pulsado el botón de "Create host" del formulario, todavía tenemos que hacer algo más para tener nuestro router disponible en Internet con el nombre elegido: debemos configurar nuestro router para que haga uso del servicio No-Ip.

La manera de conseguir esto varía según el modelo de router, pero en todo caso es una opción existente dentro del panel de control de dicho dispositivo, al cual normalmente se accede vía web especificando su ip privada en un navegador y un nombre de usuario y contraseña en el cuadro que aparece. Estos tres datos los ha de proporcionar el fabricante. Sea como sea, en el apartado de configuración DDNS nos preguntará la empresa que ofrece el servicio (en nuestro caso, No-IP), y el nombre de usuario y contraseña utilizado en la creación del nombre DNS. Una vez hecho esto, el router mantendrá una comunicación con No-IP para notificar automáticamente cualquier cambio en la vinculación ip pública<->nombre, de manera que, ahora sí, nuestro router esté siempre disponible en Internet. Nota: es posible que algún modelo de router no tenga en la lista de servicios DDNS configurables el servicio No-IP; en ese caso, o bien se ha de utilizar otro servicio similar que sí sea compatible, o bien, se puede intentar sobrescribir el firmware del router por otro que sí soporte No-IP, como por ejemplo DD-WRT (http://www.dd-wrt.com). No obstante, esta última solución solo se recomienda para usuarios que realmente sepan lo que están haciendo, ya que se puede acabar con un router inservible.

No obstante, aún falta un paso más: con el uso de No-Ip nuestro router ya es accesible a Internet, pero dentro de nuestra LAN podemos tener conectados varios equipos a él (entre los cuales, nuestra placa/shield Arduino). ¿Cómo sabe el router que cuando reciba un mensaje, este va dirigido a un equipo concreto de entre los que tiene conectados? Mediante la redirección de puertos. Este es otro apartado del panel de control del router, habitualmente señalado como "Port Forwarding", en el cual se puede reenviar mensajes que al router le llegan dirigidos a un puerto específico a otro puerto específico de una máquina concreta de la LAN, identificada por su ip privada. Es decir, si suponemos que nuestra placa/shield Arduino tiene el puerto 23 abierto (como en el ejemplo anterior), en la configuración de reenvío de puertos del router tendríamos que especificar que los mensajes recibidos por el puerto 23 del router se redirijan al puerto 23 de una máquina de nuestra LAN con ip privada 192.168.1.177 (en nuestro ejemplo). Y ahora sí que ya tendremos nuestra placa/shield abierta para escuchar peticiones de todo el mundo, literalmente. Hay que tener en cuenta, no obstante, que cada puerto solo puede ser redireccionado una sola vez.

Los servicios gratuitos de No-IP caducan en caso de inactividad: si no se accede al nombre DNS en el plazo de 30 días, el dominio será borrado del sistema. Es

posible evitar que caduque el servicio haciendo clic sobre un enlace en un mail de aviso de caducidad que es enviado tras 25 días de inactividad, o también comprando un servicio No-IP de pago.

Uso de Arduino como cliente

Si lo que queremos es que nuestra placa Arduino se conecte como cliente a cualquier otro dispositivo de red que actuará como servidor (es decir, que nuestra placa Arduino solicite algún recurso externo), lo primero que debemos hacer es declarar un objeto de tipo "EthernetClient" (en el ámbito de nuestro sketch donde se desee), que representará la propia placa. Suponiendo que este objeto EthernetClient lo llamamos "micliente", lo siguiente que debemos hacer es conectarnos al servidor que deseemos. Esto se hace con la siguiente instrucción:

micliente.connect(): realiza la conexión con el servidor cuya ip se haya especificado como primer parámetro (también se puede especificar un nombre DNS, si previamente se escribió algún servidor DNS en *Ethernet.begin()*) y cuyo número de puerto se haya especificado como segundo parámetro. Si se especifica una ip, esta ha de ser un array de tipo "byte" de cuatro elementos previamente declarado e inicializado. Si se especifica un nombre DNS, este es simplemente una cadena de caracteres. El valor de retorno de esta función es "true" si la conexión se ha realizado con éxito o "false" en caso contrario.

Queda claro de la función anterior que para realizar una conexión con un servidor externo, nuestra placa/shield Arduino ha de conocer, además de su ip, un número de puerto adecuado. No vale escribir uno cualquiera, hay que especificar el número de puerto concreto por el que ese servidor admite peticiones, ya que todas las peticiones dirigidas a un puerto diferente del que pone a disposición el servidor serán ignoradas. Por suerte, hay una serie de números de puerto estandarizados que suelen utilizarse siempre para las mismas tareas; así, por ejemplo, los servidores HTTP (también conocidos como servidores "web", es decir, computadores que ofrecen páginas web para que los clientes HTTP –los llamados "navegadores"– puedan visitarlas) suele utilizar el puerto 80. Por lo tanto, si nuestra placa Arduino se ha de conectar a un servidor HTTP, deberemos escribir su ip y el puerto 80 en micliente.conect().

Otros ejemplos de servidores son los servidores FTP (servidores que permiten la transferencia –subida y bajada– de ficheros con clientes FTP), los cuales tienen

abierto el puerto 21. O los servidores SSH (servidores que permiten el acceso remoto desde clientes SSH al terminal de comandos), los cuales tienen abierto el 22. También existen los servidores Telnet (similares a los SSH pero menos seguros, ya que no cifran la información transmitida), que abren el puerto el 23. Y los servidores SMTP (usados por los clientes SMTP para enviar correos electrónicos a su destino) y los servidores POP3 (usados por los clientes POP3 para leer los correos electrónicos de nuestro buzón), que utilizan el puerto 25 y 110, respectivamente. Y así hasta miles y miles de servidores diferentes. Si se quiere conocer la lista estandarizada completa, se puede consultar http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.txt.

Para enviar datos desde "micliente" al servidor al que estemos conectados, se pueden usar las funciones:

micliente.print(): su comportamiento es exactamente igual al de *micliente.print()* visto en el apartado anterior, pero esta vez la transmisión de los datos va desde la placa Arduino actuando como cliente hacia un servidor exterior.

micliente.println(): su comportamiento es exactamente igual al de *micliente.println()* visto en el apartado anterior, pero esta vez la transmisión de los datos va desde la placa Arduino actuando como cliente hacia un servidor exterior.

micliente.write(): su comportamiento es exactamente igual al de *micliente.write()* visto en el apartado anterior, pero esta vez la transmisión de los datos va desde la placa Arduino actuando como cliente hacia un servidor exterior.

Para recibir datos provenientes del servidor (normalmente, una respuesta a la solicitud previa enviada por el cliente), podemos usar:

micliente.available(): devuelve el número de bytes disponibles para leer (devolverá 0 si no hay ninguno, lógicamente) provenientes del servidor. Es decir, la cantidad de datos que han sido enviados desde el servidor a "micliente". No tiene parámetros.

micliente.read(): su comportamiento es exactamente igual al de *micliente.read()* visto en el apartado anterior, pero esta vez los datos son recibidos por la placa Arduino actuando como cliente provenientes de un servidor exterior.

micliente.flush(): elimina todos los bytes que han llegado al buffer de entrada de “micliente” provenientes del servidor externo que no han sido leídos. No tiene ni parámetros ni valor de retorno.

Finalmente, disponemos de dos funciones de control más:

micliente.stop(): desconecta “micliente” del servidor externo.

micliente.connected(): Devuelve “true” si “micliente” está conectado o “false” en caso contrario. Muchas veces es el servidor el que cierra la conexión, y esta función nos sirve para detectar si esto ha ocurrido. En ese caso lo más normal es cerrar esa conexión también por nuestra parte, mediante *micliente.stop()*. Observar que se considera que un cliente está conectado aunque la conexión haya sido cerrada si todavía existen datos sin leer. Esta función no tiene parámetros.

Ejemplo 8.2: Con un ejemplo de código se ve todo más claro. En este sketch, nuestra placa/shield Arduino se conecta al buscador de Google (por tanto, a un servidor web escuchando en el puerto 80), y le envía una cadena de caracteres determinada que representa una petición de búsqueda, concretamente de páginas web que contengan la palabra “arduino”. Podremos ver por el “Serial monitor” la respuesta que el buscador de Google nos devuelve. Para que funcione este ejemplo (no muy práctico, pero sí muy ilustrativo), solo necesitamos que la placa tenga acceso a una puerta de enlace funcional en nuestra LAN (normalmente a través de la conexión a un switch de red) con ip 192.168.1.1. La placa también ha de estar conectada vía USB a nuestro computador.

```
#include <Ethernet.h>
#include <SPI.h>
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
byte ip[] = { 192, 168, 1, 177 };
//Una ip del servidor web de Google (en concreto, su buscador)
byte miservidor[] = { 173, 194, 67, 103 };
EthernetClient micliente;
void setup() {
       Ethernet.begin(mac, ip);
       Serial.begin(9600);
       /* Si micliente.connect() devuelve 0, la conexión no se ha
       podido realizar. Los servidores web siempre escuchan en el
       puerto 80 */
       if (micliente.connect(miservidor, 80) != 0) {
           Serial.println("Conectado");
```



```
        /*Se envía esta cadena de caracteres al buscador de
        Google.Veremos su significado enseguida */
            micliente.println("GET /search?q=arduino HTTP/1.1");
            micliente.println("Host: www.google.com");
            micliente.println();//Línea en blanco:marca de final
        }
}
void loop(){
       char c;
       /*Si hay datos disponibles que han llegado
       desde el servidor pendientes de leer*/
        if (micliente.available() > 0) {
                //...se lee un byte en cada repetición del loop
               c = micliente.read();
               /*...y se muestra ese byte por el "Serial monitor". El
               resultado final será      la respuesta del servidor
               Google a la cadena que se ha enviado en setup()*/
               Serial.print(c);
        }
       //Si el servidor de Google ha cerrado la conexión
        if (micliente.connected() == 0) {
               micliente.stop();   //La cerramos nosotros también
              for (;;){;} //Y no hago nada nunca más
        }
}
```

¿Qué significado tiene la cadena de caracteres "*GET /search?q=arduino HTTP/1.0*" enviada en el código anterior al buscador de Google? ¿Y la línea "`Host: www.google.com`"? Hemos dicho que lo que hacíamos era una petición web para obtener las páginas que contienen la palabra "arduino", pero ¿por qué tiene esa sintaxis? Porque es una petición HTTP.

Breve nota sobre el protocolo HTTP:

El protocolo HTTP es un idioma básico formado por preguntas y respuestas que todos los clientes web (es decir, los navegadores) y los servidores web (los programas que ofrecen las páginas web a todo el mundo) entienden. Aunque se utilicen diferentes clientes web (Firefox, Chrome, Safari, Internet Explorer… o en este caso, la propia placa Arduino) y en "el otro lado" esté funcionando distinto software de servidor web (Apache, Nginx, IIS, etc.), el protocolo HTTP es estándar y permite la comunicación entre ambos extremos de una forma universal.

Básicamente, el protocolo HTTP consta de solicitudes concretas que puede realizar un cliente web, y de respuestas predefinidas que puede ofrecer un servidor web a estas solicitudes. De entre los tipos de solicitudes posibles, la más habitual con diferencia es la de solicitar una página web; esta solicitud se efectúa enviando el comando "GET" al servidor; y tras él, el nombre de la página concreta que queremos obtener de ese servidor web.

En el código de ejemplo anterior, el símbolo "/" significa que la página solicitada es la página inicial del sitio web. Y la cadena que sigue a continuación (`search?q=arduino`) es el parámetro que se le pasa a esta página inicial para que sea interactiva (en este caso, para que busque la palabra "arduino"). De hecho, podemos comprobar fácilmente cómo, si accedemos a la página del buscador de Google a través de un navegador normal, dependiendo de lo que escribimos en el cuadro de texto, así se modifica la cola "search?q=" de la dirección visible en la barra de direcciones. Finalmente, la cadena `HTTP/1.1` especifica la versión del protocolo que el cliente web está usando (en este caso, la 1.1, que es la más moderna); esta cadena siempre se ha de poner al final de la orden GET.

Si hubiéramos querido hacer una petición GET a otra página diferente que no fuera la principal del sitio (por ejemplo a http://arduino.cc/en/Reference/HomePage en vez de a http://arduino.cc), tendríamos que añadir tras la barra "/" toda la cadena tras la dirección del servidor web. Es decir, en el ejemplo anterior la petición GET debería ser `GET /en/Reference/HomePage HTTP/1.1`

Normalmente, las solicitudes HTTP no se componen solamente de una línea GET sino que están formadas por más líneas, que si no se especifican, toman un valor por defecto. Estas líneas (que en conjunto se denomina "la cabecera" de la solicitud cliente) sirven para informar al servidor sobre detalles más técnicos de la petición. En el código anterior se envía una línea de cabecera concreta, que es la línea "Host: xxxx", donde xxxx representa siempre el nombre DNS del servidor web al que el cliente quiere conectar. Esta línea es imprescindible indicarla si se utiliza la versión 1.1 de HTTP, así que normalmente siempre veremos una petición GET acompañada de una línea de cabecera "Host:xxxx".

Las respuestas HTTP enviadas por el servidor tras una solicitud hecha por el cliente pueden ser variadas: si el servidor puede ofrecer con éxito lo que el cliente pedía, el servidor se lo indica enviándole un código de respuesta tal como "HTTP/1.1 200 OK", donde en esa cadena se especifica la versión del protocolo usado y un código numérico estándar prefijado que identifica el significado de esa respuesta (además de un texto más entendible). Cada tipo de respuesta tiene un código numérico

prefijado que indica el tipo de respuesta. Por ejemplo, el famoso 404 que aparece cuando accedemos a una página no existente, es precisamente el código HTTP que el servidor envía al cliente informando de esta circunstancia. Para conocer el conjunto de códigos numéricos de respuesta del servidor, podemos consultar el documento oficial del estándar HTTP en http://www.ietf.org/rfc/rfc2616.txt.

Igualmente pasa en las solicitudes de los clientes, las respuestas de los servidores no solamente se componen de la línea con el código identificador del tipo de respuesta y ya está, sino que están formadas por más líneas específicas del protocolo HTTP. Estas líneas (que en conjunto se denominan "la cabecera" de la respuesta del servidor) sirven para informar al cliente sobre detalles más técnicos de la respuesta.

Otra duda que podemos tener ejecutando el código anterior es el significado de lo que visualizamos por el "Serial monitor". Hemos dicho que es la respuesta que nos ofrece Google en base a la petición que le hemos hecho, pero ¿de qué consta esa respuesta? Consta de dos partes: las primeras líneas son la cabecera de la respuesta del servidor. Y, tras una línea vacía (en blanco), empieza el envío propiamente dicho del contenido solicitado por el cliente (es decir, de la página web). En este caso sería la primera página web con el listado de resultados ofrecido por Google tras clicar en el botón de "Buscar". Lógicamente, si la solicitud la realizamos a través de un navegador, lo que veremos es esa página, pero si la solicitud la realizamos mediante la placa/shield Arduino, el resultado visible por el "Serial monitor" no se parece en nada. ¿Por qué? Porque lo que estamos viendo en realidad es el código HTML de la página.

Breve nota sobre el lenguaje de marcas HTML:

Todas las páginas web están escritas básicamente en un "lenguaje" llamado HTML. Cuando un navegador solicita una página web, lo que recibe del servidor es precisamente ese código fuente en HTML (que es lo que vemos por el "Serial Monitor". La función del navegador es saber interpretar ese código HTML en elementos visibles para el usuario (imágenes, párrafos, títulos, etc.). Gracias a que el lenguaje HTML es estándar, da igual –en principio– qué navegador utilicemos porque la página se debería visualizar correctamente.

El HTML no es un lenguaje de programación realmente, sino un simple conjunto de etiquetas que indican dónde se tiene que visualizar una fotografía o un texto; lo que se llama un "lenguaje de marcas". Se sale de los objetivos del libro el estudio

de este estándar, pero si se quiere empezar a conocerlo, recomiendo los estupendos tutoriales presentes en http://www.w3schools.com/html y www.w3schools.com/html5.

Ejemplo 8.3: Podemos modificar el código anterior para que en vez de que siempre se realice la misma búsqueda (la palabra "arduino"), podamos decidir cada vez que se inicia la ejecución del sketch la frase a buscar en Google. Para ello, deberemos introducir esa frase mediante el "Serial monitor".

```
#include <Ethernet.h>
#include <SPI.h>
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
byte ip[] = { 192, 168, 1, 177 };
byte servdns[] = { 8, 8, 8, 8 };
//Usamos su nombre DNS en vez de su ip
char servidor[] = "www.google.com";
EthernetClient micliente;
//Frase a buscar
String frase;
byte i=0;
void setup() {
      char c;
       Ethernet.begin(mac, ip, servdns);
       Serial.begin(9600);
        if (micliente.connect(servidor, 80) != 0) {
             Serial.println("Conectado");
//Mientras no se escriba la frase a buscar, no se hace nada
            while (Serial.available() == 0) {;}
/*Arduino lee carácter a carácter lo que se ha escrito en el
"Serial monitor" y los guarda en un String */
            while (Serial.available() > 0){
/*Es importante que "c" sea de tipo "char" y no "byte" para
que en la frase no se guarde el código numérico ASCII sino el
carácter en sí*/
                   c=Serial.read();
//Añadimos el carácter leído al final de la frase
                   frase.concat(c);
/*Este delay es muy importante para darle tiempo a Serial.available()
a recibir el siguiente carácter en el buffer. Si no se pone, el
       código
Arduino es más rápido que la llegada de nuevos bytes y la función
Serial.available() puede devolver 0 cuando todavía no se han recibido
todos los bytes */
```



```
                    delay(100);
               }
         } else {
               Serial.println("NO Conectado");
         }
         if (micliente.connected() !=0 )  {
             micliente.print("GET /search?q=");
             micliente.print(frase);
             micliente.println(" HTTP/1.1");
             micliente.println("Host: www.google.com");
             micliente.println();
         }
}
void loop(){
      char c;
//Si recibo respuesta del servidor...
         if (micliente.available() > 0) {
//La muestro carácter a carácter en el "Serial monitor"
              c = micliente.read();
             Serial.print(c);
          }
          if (micliente.connected() == 0) {
             Serial.println("Desconexión");
               micliente.stop();
             for(;;){;}
          }
}
```

Caso práctico: servidor web integrado en la placa/shield Arduino

En el siguiente código, convertimos nuestra placa/shield en un servidor web simple. Es decir, hacemos que nuestra placa/shield pueda contestar a peticiones HTTP para ofrecer a los clientes contenido HTML. Por tanto, desde un computador cualquiera podríamos abrir un navegador y escribir en su barra de direcciones la ip privada de la placa/shield (si estamos dentro de una LAN) o su nombre DNS (si conectamos a través de Internet) para así contactar con ella y visualizar el contenido HTML ofrecido por esta en forma de página web. En este caso concreto, supondremos que tenemos conectado a la placa/shield un sensor analógico cualquiera (un LDR, un termistor, etc.) cuyas lecturas serán visibles en esa página web Si no tenemos ningún sensor a mano, el código funcionará perfectamente, solo que los valores leídos serán ruido aleatorio.

```
#include <SPI.h>
#include <Ethernet.h>
byte mac[] = {0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
IPAddress ip(192,168,1,177);
EthernetServer miservidor(80);
EthernetClient micliente;
//Sirve para saber cuándo acaba la solicitud recibida del cliente
boolean lineaActualEstaVacia = true;
/*Cada carácter recibido del cliente
que forma parte de la petición HTTP */
char caracter;
/*Pin de la placa Arduino donde se supone
que está conectado el sensor analógico*/
byte pinAnalogico = 0;
//Valor devuelto de la lectura del sensor analógico
int lectura=0 ;
void setup() {
  Ethernet.begin(mac, ip);
  miservidor.begin();
  /*Para ver la solicitud enviada del cliente
  al servidor (las cabeceras de la petición)*/
  Serial.begin(9600);
}
void loop() {
  micliente = miservidor.available();
  //Si se detecta una conexión de algún cliente
  if (micliente == true) {
    Serial.println("Nueva conexión de cliente");
    while (micliente.connected() == true) {
      //Procedo a leer carácter a carácter la petición HTTP
      if (micliente.available() > 0) {
        caracter = micliente.read();
//Por el "Serial monitor" veo las líneas de cabecera de la petición
        Serial.print(caracter);
        /*Si el carácter recibido es un nuevo \n (salto de línea)
        significa que hemos llegado al final de esa línea. Y si
        además esa lineaActualEstaVacia es true, significa que el
        cliente nos ha enviado una línea entera sin caracteres,
        completamente vacía (en blanco). Una línea completamente
        vacía es la señal estándar para indicar que la petición HTTP
        ha acabado, y por tanto, que podemos enviar una respuesta. */
        if (caracter == '\n' && lineaActualEstaVacia == true) {
            /*Envío una cabecera HTTP estándar. Esta consta de la
          línea con el código de respuesta 200 (OK), otra línea
```

```
		llamada Content-Type: que indica al cliente el tipo de
		página web que se le va a enviar (normalmente, siempre es
		"text/html") y la línea llamada Connection:, cuyo valor
		"close" indica que el servidor cerrará la conexión con el
		cliente una vez enviada la página web solicitada (es decir,
		no realiza una conexión persistente).*/
		micliente.println("HTTP/1.1 200 OK");
		micliente.println("Content-Type: text/html");
		micliente.println("Connnection: close");
		/*Una línea en blanco marca el fin del envío de las
		cabeceras HTTP. A partir de aquí se envía el contenido HTML
		propiamente dicho. En este ejemplo la placa Arduino envía
		una sola página HTML, por lo que ésta es la página web por
		defecto. Esto significa que no habrá que indicar nada
		especial en el navegador para visualizarla aparte de
		especificar la ip/nombre de la placa.*/
		micliente.println();
		/*El código HTML está formado por una serie de "etiquetas"
		que indican el tipo de contenido que se va a mostrar. Estas
		etiquetas se indican entre "<" y ">". Las dos siguientes
		líneas son dos etiquetas que obligatoriamente tienen que
		aparecer siempre al principio del código HTML de cualquier
		página*/
		micliente.println("<!DOCTYPE HTML>");
		micliente.println("<html>");
		/*Esta etiqueta indica al navegador que actualice la página
		(es decir, que vuelva a hacer una petición al servidor)
		automáticamente cada 5 segundos. Si no añadimos esta línea,
		el navegador solo obtendrá una vez el dato leído del
		sensor, y para obtenerlo cada vez deberemos hacer una
		petición nueva "a mano" */
		micliente.println("<meta http-equiv=\"refresh\"
content=\"5\">");
		/*Recojo el valor del sensor analógico y lo
		muestro dentro del código HTML que verá el navegador */
		lectura = analogRead(pinAnalogico);
		micliente.print(lectura);
		/*La etiqueta "<br />" sirve para añadir
		un salto de línea visible en una página web. */
		micliente.println("<br />");
		/*Todo código HTML ha de finalizar con la etiqueta
		"</html>", que además marca el fin del contenido enviado
		por el servidor */
		micliente.println("</html>");
```

```
          /*Ya no hay nada más que enviar. Salimos del bucle
          y ya fuera cerraremos la conexión */
          break;
        }
        //Acabo una línea y empiezo una nueva, en principio vacía
        if (caracter == '\n') {
          lineaActualEstaVacia = true;
        /*El carácter leído es uno cualquiera de esa línea, por lo
        que no está vacía. La excepción del carácter \r (retorno de
        carro) proviene de que \r siempre precede a \n en todas sus
        apariciones, por lo que en realidad, una línea vacía siempre
        tiene el carácter \r. Así que en el caso de detectar el
        carácter \r, la línea se sigue considerando vacía. Es decir,
        básicamente lo que se está buscando es una secuencia así:

caracteres de una línea, incluyendo \r --> lineaActualEstaVacia=false
\n (salto de línea)                    --> lineaActualEstaVacia=true
\r (carácter "fantasma", no pasa nada) --> lineaActualEstaVacia=true
\n (salto de línea) + lineaActualEstaVacia=true -> Envío respuesta*/

        } else if (caracter != '\r' ) {
          lineaActualEstaVacia = false;
        }
      }
    }
    //Damos tiempo al navegador a recibir los datos
    delay(1);
    /*Como ya hemos enviado la página al cliente, no tenemos
    nada más que hacer con él, así que cerramos la conexión */
    micliente.stop();
  }
}
```

Caso práctico: servidor web con tarjeta SD

Supongamos que tenemos un fichero llamado “pagina.html” almacenado dentro de una tarjeta SD conectada a nuestra placa/shield Ethernet. En el ejemplo anterior hemos visto cómo servir una página web generada en tiempo real, pero ¿cómo podemos servir una página web que ya esté guardada previamente dentro de la tarjeta SD? Pues de una forma muy similar a la que hemos visto. A continuación, se presenta un código de ejemplo, donde lo único que cambia respecto el código anterior es la respuesta que el servidor ofrece al cliente tras enviarle las pertinentes cabeceras HTTP.

```
#include <SPI.h>
#include <Ethernet.h>
#include <SD.h>
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
byte ip[] = { 192,168,1, 177 };
File pagina;
EthernetServer miservidor(80);
EthernetClient micliente;
char caracter;
boolean lineaActualEstaVacia = true;
void setup(){
       Ethernet.begin(mac, ip);
       miservidor.begin();
        pinMode(10,OUTPUT); //Necesario para la librería SD
       /*Si ha habido un error en inicializar
       la tarjeta, se aborta el programa */
       if (!SD.begin(4)) { return; }
}
void loop() {
  micliente = miservidor.available();
  if (micliente == true) {
    while (micliente.connected()== true) {
      if (micliente.available() > 0) {
        caracter = micliente.read();
        if (c == '\n' && lineaActualEstaVacia==true) {
          client.println("HTTP/1.1 200 OK");
          client.println("Content-Type: text/html");
          client.println();
      /*Tras el envío de las cabeceras HTTP se abre el fichero
       almacenado en la tarjeta SD y se va leyendo carácter tras
       carácter. Al mismo tiempo que se lee cada carácter, este se
       reenvía por la red hacia el cliente.*/
          pagina = SD.open("pagina.html");
          if (pagina != 0) { //Si no hay error…
            //Mientras haya caracteres por leer en el fichero…
            while (pagina.available()> 0) {
                //…los reenvío al cliente
                micliente.write(pagina.read());
            }
           //Después de leerse todos los caracteres,cierro el fichero
            pagina.close();
          }
          break; //Ya he enviado todo. Salgo para cerrar la conexión
        }
```

```
          if (caracter == '\n') {
            lineaActualEstaVacia = true;
          } else if (caracter != '\r') {
            lineaActualEstaVacia = false;
          }
        }
      }
      delay(1);
      micliente.stop();
    }
}
```

El código anterior suele ser utilizado sobre todo para mostrar páginas web cuyo contenido está formado por la información obtenida periódicamente de algún sensor y grabada a tiempo real en la tarjeta SD. Como ejemplo de código donde se ve cómo se almacenan en una tarjeta SD los datos recibidos por un sensor podemos recordar el estudiado en este libro dentro del apartado de sensores de humedad, así que no recomiendo su relectura. En aquel ejemplo, los datos se guardaban en simples ficheros de texto, pero en el caso de quererlos guardar en páginas web el procedimiento es muy similar, porque en realidad, las páginas web no son más que ficheros de texto escritos usando unas determinadas reglas (más concretamente, usando unas determinadas etiquetas HTML). Por tanto, si escribimos esas etiquetas convenientemente, podemos almacenar la información de manera que generemos una página web estándar (visible por cualquier navegador), en vez de generar un simple fichero de texto tabulado.

Al código anterior también le podríamos haber añadido la capacidad de ofrecer diferentes ficheros según determinadas circunstancias (la pulsación de un botón, la lectura recibida de un sensor, etc.). Se deja como ejercicio.

Otra opción que se deja como ejercicio es hacer que el usuario conectado a nuestra placa/shield Ethernet visualice una página web que muestre un menú de enlaces, cada uno correspondiente a un fichero guardado en la tarjeta SD; de esta manera, el usuario podría elegir en cada momento qué fichero quiere descargar. El código que permite esta posibilidad es ciertamente algo más complejo y largo de lo que estamos mostrando en este libro, pero no por ello deja de ser interesante su estudio. Lo podemos obtener de aquí: https://github.com/adafruit/SDWebBrowse.

Caso práctico: formulario web de control de actuadores

En el ejemplo del apartado anterior, hemos supuesto que teníamos una serie de sensores conectados a nuestra placa/shield Arduino y que lo que queríamos era acceder a los valores leídos por ellos a través de una página web ofrecida por la propia placa. Pero ¿y si lo que queremos es utilizar esa página web no para observar datos de sensores sino para controlar el comportamiento de actuadores? Es decir, para poder mover motores o encender LEDs simplemente apretando un botón visible dentro de nuestra web. ¿Qué tendríamos que hacer?

Pues algo muy parecido al ejemplo anterior: tenemos que montar un servidor web en la placa/shield Arduino que acepte peticiones de computadores clientes, los cuales verán una simple página web ofrecida por ella. Lo que cambia ahora básicamente es el código HTML de esa página, ya que en vez de mostrar un valor y ya está (como hacía en el ejemplo anterior), esta vez la página tiene que mostrar una serie de botones que han de reaccionar de una manera determinada cuando son pulsados (moviendo un motor en un sentido o en otro, acelerándolo o frenándolo, encendiendo o apagando un LEDs, emitiendo una melodía con un zumbador u otra, etcétera).

El siguiente código se ofrece como plantilla de referencia a partir de la cual se puede extender y ampliar su funcionalidad para introducir nuevos actuadores. Tal como se presenta, mueve un servomotor conectado al pin de salida PWM número 6 de tres maneras (mediante tres botones). Cada vez que se pulse el primer botón el servomotor se moverá 20 grados en un sentido, cada vez que se pulse el segundo botón se moverá 20 grados en el otro sentido, y cada vez que se pulse el tercero el servomotor se moverá de 0 a 180 grados, y viceversa.

```
#include <SPI.h>
#include <Ethernet.h>
#include <Servo.h>
byte mac[] = {0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
IPAddress ip(192,168,1,177);
EthernetServer miservidor(80);
EthernetClient micliente;
Servo miservo;
int pos=90; //No es "byte" porque "byte" no admite negativos
boolean lineaActualEstaVacia = true;
char caracter;
String peticion="";
void setup() {
```

```
  Ethernet.begin(mac, ip);
  miservidor.begin();
  miservo.attach(6);
}
void loop() {
  int i;
  micliente = miservidor.available();
  if (micliente == true) {
    while (micliente.connected() == true) {
      if (micliente.available() > 0) {
        caracter = micliente.read();
/*Solo nos interesa almacenar los primeros caracteres de la petición
del cliente para saber qué botón se ha pulsado (hemos puesto 30 por
conveniencia, pero se puede cambiar); no nos interesa guardar toda la
petición completa */
        if (peticion.length() < 30 ){
                  peticion.concat(caracter);
        }
        if (caracter == '\n' && lineaActualEstaVacia==true) {
          micliente.println("HTTP/1.1 200 OK");
          micliente.println("Content-Type: text/html");
          micliente.println("Connnection: close");
          micliente.println("");
          micliente.println("<!DOCTYPE HTML>");
          micliente.println("<html>");
          /*Todo lo que hay entre la etiqueta
          <h1> y </h1> es un título*/
          micliente.println("<h1>Formulario de control de
servos</h1>");

/*Aquí comprobamos qué petición ha realizado el cliente (es decir,
qué botón se ha pulsado del navegador). Dependiendo de la opción
elegida, se mostrará en la página web una frase u otra y se moverá el
servo de la forma correspondiente. Esta zona del código se puede
ampliar para lo que se desee: encender un LED, hacer sonar un
zumbador, etc.*/
          if (peticion.indexOf("SentidoServo=0") > 0 ) {
              micliente.println("Sentido del movimiento
actual:Izquierda <br/>");
              micliente.println("Posición angular actual:");
              pos=pos+20;
              /*Si dentro del if solo hay una instrucción,
              se puede escribir sin llaves */
              if (pos > 180) pos=180;
```



```
              miservo.write(pos);
              micliente.println(pos);
          }
          if (peticion.indexOf("SentidoServo=1") > 0 ){
              micliente.println("Sentido del movimiento
actual:Derecha <br/>");
              micliente.println("Posición angular actual:");
              pos=pos-20;
              if (pos < 0) pos=0;
              miservo.write(pos);
              micliente.println(pos);
          }
          if (peticion.indexOf("SentidoServo=2") > 0 ){
              micliente.println("Sentido del movimiento actual:Vuelta
completa <br/>");
              for (i=0;i<180;i++){
                   miservo.write(i); delay(5);
              }
              for (i=180;i>0;i--){
                   miservo.write(i); delay(5);
              }
          }
/*Una vez realizadas las comprobaciones de control de servos, acabo
de enviar el resto de página que falta. Esta consta de un formulario
con los botones de control */
          micliente.println("<br/>");
          micliente.println("<form>");
          micliente.println("<button type='submit'
name='SentidoServo' value='0'>Izquierda</button>");
          micliente.println("<button type='submit'
name='SentidoServo' value='1'>Derecha</button>");
          micliente.println("<button type='submit'
name='SentidoServo' value='2'>Vuelta completa</button>");
          micliente.println("</form>");
          micliente.println("</html>");
          break;
        }
        if (caracter == '\n') {
          lineaActualEstaVacia = true;
        } else if (caracter != '\r' ) {
          lineaActualEstaVacia = false;
        }
      }
    }
```

```
/*Vacío la cadena de la petición del cliente actual, para empezar de
nuevo en la posible siguiente conexión */
    peticion="";
    micliente.stop();
  }
}
```

Lo más importante para entender perfectamente el código anterior es saber cómo funciona el mecanismo de envío de peticiones desde el navegador a la placa Arduino. En realidad, se ha empleado la manera más sencilla posible de interactuar vía web con la placa Arduino: mediante el uso de botones dentro de un formulario HTML. De esta manera, si se pulsa un botón, se envía una petición determinada, si se pulsa otro, se envía otra, y la placa responde a cada petición en consecuencia.

Un formulario HTML comienza siempre con la etiqueta *<form>* y acaba con la etiqueta *</form>*. Entre estas dos etiquetas se han de colocar los elementos que formarán ese formulario, los cuales pueden ser de muchos tipos (cajas de texto, listas desplegables, “checkboxs”, etc.). A nosotros el elemento de formulario que nos interesa es el botón de envío de datos. Por cada botón de este tipo que queramos incluir, debemos escribir una etiqueta *<button>* con una serie de “parámetros”. El parámetro “type” sirve para indicar que la función de ese botón es enviar un dato determinado a alguna página web para que esta lo procese. Esa página web debe estar definida dentro de la etiqueta *<form>* mediante el parámetro “action”, pero si este no se especifica (tal como hemos hecho nosotros), el dato entonces va destinado a la misma página web de donde surgió. Esto es lo que nos interesa porque la placa Arduino solo muestra una sola página que realiza las dos cosas: mostrar el formulario y procesar los datos enviados desde él. El parámetro “name” del botón sirve para indicar el nombre del dato que se enviará, y el parámetro “value” sirve para indicar su valor. Es decir, los datos siempre se envían por parejas nombre<->valor, de tal manera que se puede identificar fácilmente cuál es el dato a procesar, y cuál es su valor concreto en ese momento. De esta forma, podríamos por ejemplo añadir a nuestro código otros botones para controlar un LED, o un motor DC, o un zumbador, etc., que tuvieran otro nombre para diferenciarlos de los que ya controlan el servomotor, y hacer que cada uno de estos nuevos botones enviara un valor diferente (HIGH/LOW, o bien un valor analógico).

Bien, ya sabemos cómo envia los datos el formulario. Pero ¿cómo se procesan? Si no se especifica lo contrario, por defecto el tipo de envío de esos datos es mediante el sistema “GET” (aunque también se puede usar otro sistema de envío algo diferente, llamado “POST”). El tipo de envío “GET” hace que en la primera línea de cabecera de petición del cliente aparezca la pareja nombre<->valor del dato

enviado con la sintaxis siguiente: *GET /?nombre=valor HTTP/1.1* . Por tanto, una manera sencilla de comprobar si se ha pulsado un determinado botón es detectar si en la cadena anterior aparece el “nombre=valor” buscado. El lenguaje Arduino dispone para los objetos String de la instrucción “indexOf()”, que nos puede servir para esto, ya que recordemos que esta instrucción siempre devuelve un valor mayor de -1 si la cadena a buscar se encuentra dentro de otra mayor.

Hay que tener en cuenta que hemos restringido a 30 caracteres (esto es por conveniencia, se puede cambiar si se necesita) la cadena utilizada para buscar dentro de ella el dato “nombre=valor” enviado. Podríamos haber añadido la cabecera entera de la petición dentro de esa cadena, pero en ese caso nos podríamos haber encontrado con un problema. Además de estar en la primera línea, el dato “nombre=valor” puede aparecer en otras líneas de la cabecera (como por ejemplo la línea “Referer:”), con lo que en ese caso el dato se detectaría varias veces y por tanto el servomotor se movería más pasos del único realmente necesario. Para evitar este problema, a los 30 caracteres (una longitud prudencial) dejamos de llenar la cadena utilizada para buscar dentro de ella el dato enviado, con lo que conseguimos que solo contenga como mucho una sola vez el dato “nombre=valor” sin repetirse.

Se pueden construir formularios HTML más complejos que son capaces de enviar varios valores a la vez en una petición GET. En esos casos, la sintaxis de esa petición es *GET /?nombre1=valor1&nombre2=valor2 HTTP/1.1* (para un envío de dos datos) o *GET /?nombre1=valor1&nombre2=valor2&nombre3=valor3 HTTP/1.1* (para un envío de tres datos), y así. En todo caso, esto es fácil comprobarlo si nos fijamos en la barra de direcciones de nuestro navegador: allí aparece escrita la cadena concreta de parejas nombre<->valor enviada tras pulsar el botón. De hecho, sería completamente equivalente escribir la cadena nombre<->valor deseada directamente en la barra de direcciones y pulsar “Enter” que apretar el botón pertinente.

Respecto al diseño de las páginas web ofrecidas por Arduino, es evidente que no es lo más avanzado del mundo. Si queremos que el aspecto de estas mejoren, lo más recomendable es utilizar un “lenguaje” complementario del HTML llamado CSS, el cual está específicamente pensado para controlar el aspecto estético de las páginas web. Combinar código CSS con código HTML hace que este se visualice de una forma mucho más vistosa. No obstante, se sale de los objetivos de este libro profundizar en este tema. Si se desea saber más, recomiendo consultar el excelente tutorial online disponible en http://www.w3schools.com/css y http://www.w3schools.com/css3.

Para acabar este apartado, diremos que en vez de utilizar botones dentro del formulario para controlar el estado del servo, podríamos haber utilizado otros

elementos de entrada de datos, como por ejemplo una caja de texto. Si sustituimos en el código del ejemplo anterior las líneas

```
micliente.println("<button type='submit' name='SentidoServo'
value='0'>Izquierda</button>");
micliente.println("<button type='submit' name='SentidoServo'
value='1'>Derecha</button>");
micliente.println("<button type='submit' name='SentidoServo'
value='2'>Vuelta completa</button>");
```

por estas:

```
micliente.println("<input type='text' name='SentidoServo'/>");
micliente.println("<button type='submit' value='0'>Enviar </button>");
```

obtendremos un formulario con una caja de texto y un botón. En la caja de texto deberemos introducir el valor de control (“0” –movimiento en un sentido– ,”1” –movimiento en otro sentido– o ”2” –vuelta completa–) que deseemos, y el botón nos servirá para enviarlo a Arduino. Cualquier otro valor diferente de “0”, “1” o “2” no hará nada. Hay que fijarse en que el elemento HTML correspondiente a una caja de texto es el elemento *<input/>* el cual ha de tener como mínimo dos parámetros: su tipo (las cajas de texto estándar son de tipo “text”) y su nombre, que servirá a la placa Arduino para identificar a qué caja de texto pertenece el dato recibido. Evidentemente, una caja de texto no tiene parámetro “value” explícito porque este es el texto que se ha introducido en cada momento. El elemento *<button>*, por su parte, es idéntico a los ya vistos, con la salvedad de que ahora no es necesario que envíen ningún “value”, porque su única función es enviar los datos introducidos en otros elementos del formulario.

A partir de los ejemplos anteriores, ya podemos ser capaces de construir un panel de control web para gestionar cualquier elemento conectado a una placa Arduino. Por ejemplo, podríamos utilizar cajas de texto para establecer el mensaje a visualizar en una LCD, o para variar la velocidad de un motor DC, o para elegir la melodía a emitir por un altavoz, etc.

Caso práctico: envío de mensajes a Twitter.com

Twitter (http://www.twitter.com) es una plataforma gratuita de microblogging. Es decir, previo registro, uno puede crear un mensaje de como máximo 140 caracteres y hacerlo visible a todo el mundo (literalmente), aunque para que otros usuarios de Twitter puedan recibir estos mensajes automáticamente,

deberán suscribirse a ellos. La manera de leer los mensajes recibidos gracias a nuestras suscripciones puede variar: se puede acceder (previa introducción de usuario y contraseña) a través de la página web oficial de Twitter, o bien utilizando alguna aplicación específica para PC o teléfono móvil de última generación.

Podemos aprovechar este sencillo funcionamiento para hacer que nuestra placa Arduino pueda publicar mensajes utilizando una cuenta propia de Twitter y usar entonces nuestra propia cuenta personal para suscribirnos a dichos mensajes. De esta forma, en todo momento estaremos informados de todo lo que nuestra placa Arduino notifique (cuando se detecte algún evento o a intervalos regulares o cuando deseemos). Si nos preocupa la privacidad de estos mensajes emitidos por la placa, siempre se puede configurar su cuenta para que solamente los usuarios especificados puedan leerlos.

Después de crear la cuenta de nuestra placa Arduino en http://www.twitter.com/signup (y otra para nuestro uso personal), para hacer que la placa pueda publicar mensajes debemos de descargarnos la librería "Arduino Tweet" de aquí: http://arduino-tweet.appspot.com. Atención: esta librería no envía los mensajes a Twitter.com sino que los envía a su propia web, y desde allí se reenvían a Twitter.com. Es decir, se utiliza una web de terceros como intermediaria. Esto es así porque como Twitter.com tiene un mecanismo de autentificación relativamente complejo llamado OAuth, la librería "Arduino Tweet" opta por delegar a una web auxiliar todo el proceso necesario de intercambio de credenciales de concesión de acceso a una cuenta, en vez de que recaiga en la propia placa Arduino. De esta manera, ganamos facilidad de uso y no saturamos de trabajo a nuestra placa Arduino. Pero por el contrario dependemos de un servicio de terceros, el cual ya nos informa de que no acepta más de un envío de mensajes por minuto (para no sobresaturarse). Si no se desea utilizar este sistema y en cambio se quiere recaer todo el proceso OAuth en la propia placa Arduino para evitar intermediarios, existe una librería Arduino con soporte para OAuth (http://www.markkurossi.com/ArduinoTwitter), pero ciertamente, su instalación y uso es mucho más complejo.

Una vez tengamos instalada la librería "Arduino Tweet", antes de poderla usar hay que dar un paso previo. Hay que conseguir un "token". Un "token" no es más que un conjunto de caracteres que servirán para autorizar a la web http://arduino-tweet.appspot.com el acceso a la cuenta Twitter de nuestra placa Arduino. Ese "token" lo deberemos de escribir dentro de nuestro sketch para que funcione. Para conseguirlo primero debemos loguearnos en Twitter con la cuenta de nuestra placa Arduino y seguidamente clicar en el enlace señalado como "Step 1" de la web http://arduino-tweet.appspot.com y seguir los pasos que vayan apareciendo.

Respecto al tema de la seguridad usando la librería "Arduino Tweet", no hay que preocuparse, porque la web intermediaria no es capaz de conocer el nombre de usuario ni contraseña de la cuenta de Twitter de nuestra placa Arduino: el propio mecanismo OAuth lo impide. En todo caso, si después de probarlo no queremos que http://arduino-tweet.appspot.com pueda seguir accediendo a la cuenta de la placa Arduino, siempre podemos quitarle ese privilegio iniciando sesión con esa cuenta y yendo al panel de control http://twitter.com/settings/connections.

El siguiente código envía un mensaje a Twitter (siempre el mismo) cada vez que la placa Arduino se reinicia. Hay que tener en cuenta, no obstante, que Twitter rechaza mensajes iguales consecutivos (error 403), por lo que seguramente, a pesar de ejecutar este código varias veces, solo saldrá el mensaje la primera vez si no cambiamos su texto.

```
#include <SPI.h>
#include <Ethernet.h>
#include <Twitter.h>
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
byte ip[] = { 192,168,1,177};
byte servdns[] = { 8,8,8,8}
byte gateway[] = { 192.168.1.1 };
byte subnet[] = { 255, 255, 255, 0 };
//Hay que sustituir  xxxx con el token de la cuenta Arduino
Twitter twitter("xxxx");
void setup() {
       Ethernet.begin(mac, ip, servdns,gateway,subnet);
       Serial.begin(9600);
       tweet("Hola");
}
void loop() {}
void tweet(char msg[]){
       boolean enviado=false;
       int codigorespuesta;
       //Conecto con Twitter para enviarle el mensaje
       enviado=twitter.post(msg);
       //Si se ha contactado con Twitter…
       if (enviado == true) {
/*…espero hasta que se haya enviado el mensaje correctamente o haya
ocurrido un error. En todo caso, obtengo la respuesta del envío en
forma de código HTTP de cabecera de servidor. El parámetro "&Serial"
es para retransmitir esta respuesta al canal Serie; si eso no se
necesita, se ha de escribir simplemente twitter.wait(); */
              codigorespuesta = twitter.wait(&Serial);
```



```
              if (codigorespuesta == 200) {
                    Serial.println("OK.");
              } else{
                    Serial.print("Algo falló ");
                    Serial.println(status);
              }
         } else{
              Serial.println("Error al contactar con Twitter.");
         }
}
```

A partir de aquí es sencillo escribir un sketch que reaccione a diversos eventos y envíe los mensajes correspondientes. Para simular estos eventos, supondremos que tenemos por ejemplo dos pulsadores: presionando uno se enviará un mensaje y presionando el otro se enviará un mensaje diferente. Lo único que deberíamos modificar entonces del código anterior es la función “setup()” para indicar los pines de señal de entrada, y la función “loop()” tal como se muestra a continuación: el resto de código permanecerá intacto.

```
void setup(){
  Ethernet.begin(mac, ip, gateway, subnet);
  Serial.begin(9600);
  pinMode(6, INPUT);
  pinMode(7, INPUT);
}
void loop(){
       char msg1[]="Ocurrió el evento 1";
       char msg2[]="Ocurrió el evento 2";
       if (digitalRead(6)==HIGH) {
              tweet(msg1);
              delay(2000); //Damos tiempo al envío y recepción
       }
       if (digitalRead(7)==HIGH) {
              tweet(msg2);
              delay(2000);
       }
}
```

Caso práctico: envío de datos a Cosm.com

Cosm.com (http://www.cosm.com), anteriormente conocido como Pachube, es una plataforma que permite a diferentes aplicaciones (escritas en diferentes lenguajes, como Java, C, Processing…) conectar a un almacén de datos online además

de a un sistema de presentaciones gráficas online. Utilizando los procedimientos adecuados, esas aplicaciones pueden guardar información en Cosm.com cada cierto tiempo y visualizarla en forma de gráficos estadísticos. En la práctica, esta plataforma se suele utilizar para monitorizar sistemas de sensores geolocalizados, vinculados a una cuenta de usuario de Cosm.com. Una cuenta gratuita de Cosm.com permite hasta 10 sensores actualizados en tiempo real, y los datos recopilados se guardan durante 3 meses.

Nuestra idea es conectar estos sensores a Arduino y conectar Arduino a Cosm.com, de manera que los sensores vayan midiendo datos en períodos definidos de tiempo y Arduino los vaya enviando automáticamente a Cosm.com para que este los guarde. De esta forma, se podrá consultar el estado actual de los sensores y diversas estadísticas más (presentadas de diferentes formas muy accesibles) desde cualquier lugar del mundo accediendo a su web (o a través de aplicaciones específicas para teléfonos móviles de última generación) mediante la cuenta de usuario vinculada a esos sensores.

Además, esta información alojada en Cosm.com también puede compartirse y ser utilizada por otras aplicaciones (a las que previamente se les ha concedido el acceso) como entrada de datos para sus procedimientos. Esto nos permitiría, por ejemplo, controlar actuadores según la información recibida de Cosm.com, actuadores que pueden estar ubicados geográficamente en un lugar muy distante de donde están los sensores.

Para enviar información a Cosm.com, no necesitamos ninguna librería Arduino en especial: tan solo debemos enviar al servidor “api.cosm.com” una petición HTTP con unas cabeceras específicas. Pero para que estas cabeceras sean aceptadas, además de tener una cuenta propia en Cosm.com, debemos obtener un par de datos (un “Api Key” y un “feedID”) que hemos de incluir siempre en nuestros sketches para que la placa Arduino esté autorizada a enviar datos a esa cuenta en particular. Para conseguir estos datos, lo primero que debemos hacer es iniciar sesión en la web de Cosm.com y crear un nuevo “dispositivo” de tipo Arduino mediante el enlace pertinente. A ese dispositivo le deberemos dar un nombre.

Una vez creado el dispositivo, debemos ir al enlace “Keys” para obtener la “Api Key” y el “feedID”. El primero es una cadena que nos identifica dentro de Cosm.com y que permitirá, al especificarla dentro de nuestros sketches, que estos tengan permiso para enviar datos a nuestra cuenta. El segundo es un número que identifica un flujo de datos particular proveniente de nuestro dispositivo y que permitirá, al especificarlo dentro de nuestros sketches, que estos puedan informar a

Cosm.com sobre qué sensor concreto es el que está realmente enviando los datos. Para un mismo dispositivo podemos definir varios “feedID”, cada uno de ellos correspondiente a un sensor diferente, pero esto no suele ser habitual, porque en realidad, con un solo “feedID”, es posible enviar los datos de varios sensores en conjunto, tal como veremos en los siguientes ejemplos.

Desde el panel de control central (accesible mediante el enlace “Console”) se pueden modificar las características del dispositivo recién creado. Se puede, por ejemplo, añadir información de geolocalización, añadir alertas (“triggers”) cuando los valores obtenidos se salgan de un determinado rango fijado, hacer que los datos recopilados sean públicos, exportar los datos a diferentes formatos, añadir etiquetas (“tags”) para ayudar a encontrar ese dispositivo entre todos los del resto del mundo según diferentes categorías, etc. También se pueden observar los datos públicos de otros usuarios de Cosm.com en https://cosm.com/feeds?status=live.

Imaginemos ahora que tenemos nuestra placa/shield Arduino Ethernet conectada a un sensor cualquiera (de luz, de temperatura, etc.) y además, mediante una puerta de enlace de nuestra LAN, a Internet. Si queremos enviar cada 10 segundos (por ejemplo) los datos de ese sensor a una cuenta particular de Cosm.com, deberíamos escribir un código parecido al siguiente:

```
#include <SPI.h>
#include <Ethernet.h>
const char APIKEY[] =”La APIKEY de api.cosm.com se ha de poner aquí”;
//Aquí se ha de poner nuestro “feed ID”
const char FEEDID[]=”00000”;
//Aquí se ha de poner el nombre del proyecto
const char USERAGENT[]=”Mi proyecto”;
byte mac[] = {0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED};
IPAddress ip(192,168,1,177);
EthernetClient micliente;
/*Usando la ip del servidor “api.cosm.com”, se reduce
  el tamaño de nuestro sketch en la memoria Flash */
IPAddress miservidor(216,52,233,121);
//char miservidor[] = "api.cosm.com";
//Cuándo se conectó a Cosm.com la ultima vez, en milisegundos
unsigned long ultimaConexion = 0;
//Estado de la conexión en la última repetición del loop
boolean conectado = false;
//Tiempo de espera entre conexiones a Cosm.com
const unsigned long intervalo = 10*1000; //10000 milisegundos
void setup() {
```

```
      if (Ethernet.begin(mac) == 0) {
          Serial.println("DHCP ha fallado. Pruebo ip fija”);
          Ethernet.begin(mac, ip);
      }
}
void loop() {
  int lectura;
/*Los datos a enviar a Cosm.com son cadenas que siempre tienen el
mismo formato: “nombredelsensor,valordelsensor”. En nombre del sensor
podemos escribir lo que queramos*/
  String mensaje= “sensor1,”;
  //Leo un sensor cualquiera, en este caso analógico
  lectura = analogRead(0);
  //Concateno el valor de la lectura al mensaje, para completarlo
  mensaje.concat(lectura);
/*Se pueden añadir múltiples lecturas de diferentes sensores en un
mismo mensaje, poniendo cada pareja nombresensor<->valorsensor en una
línea diferente. Cada una de ellas aparecerá en Cosm.com como un
“Datastream” diferente dentro del “feed” utilizado. Por ejemplo:
  int otralectura = analogRead(1);
  mensaje.concat("\nsensor2,");
  mensaje.concat(otralectura);   */
/*Si no se está conectado a Cosm.com actualmente y ya han pasado 10
segundos desde la última conexión, vuelvo a conectar y envío el
dato*/
  if(micliente.connected()==false && (millis() - ultimaConexion > intervalo))
  { envioDatos(mensaje);
  }
}
void envioDatos (String dato) {
  if (micliente.connect(miservidor, 80)==true) {
/*Envío una petición HTTP al servidor web “api.cosm.com”. En este
caso, la petición no es de tipo GET (no queremos ver ninguna página
web) sino de tipo PUT (porque queremos enviarle unos datos para que
los recoja). Las otras líneas de la cabecera han de ser tal como se
ponen */
      micliente.print("PUT /v2/feeds/");
      micliente.print(FEEDID);
/*Los datos a enviar son valores separados por comas. Esto se conoce
como formato CSV (“comma-separated values”), pero también podríamos
utilizar otros, como JSON o XML*/
      micliente.println(".csv HTTP/1.1");
      micliente.println("Host: api.cosm.com");
      micliente.print("X-ApiKey: ");
```



```
        micliente.println(APIKEY);
        micliente.print("User-Agent: ");
        micliente.println(USERAGENT);
        micliente.print("Content-Length: ");
        //He de enviar el número de bytes que ocupa el mensaje
        micliente.println(dato.length());
        micliente.println("Content-Type: text/csv");
        micliente.println("Connection: close");
        micliente.println(); //Fin de la cabecera de cliente
       /*Ahora es cuando realmente se envía el mensaje con
       los datos de la petición PUT */
        micliente.println(dato);
        delay(100); //Me espero un momento y me desconecto
        micliente.stop();
   } else { //Si no puedo conectar con éxito, me desconecto
        micliente.stop();
   }
   //Anoto cuándo la conexión se realizó o intentó
   ultimaConexion = millis();
}
```

El código anterior nos sirve como muestra para saber cómo se podría implementar un cliente web estándar que solicitara una misma página cada diez segundos (siempre que cambiemos el tipo de petición PUT por una petición GET y hagamos unos mínimos cambios más). Se deja como ejercicio.

En realidad, utilizando la librería Arduino oficial de Cosm.com (https://github.com/cosm/cosm-arduino) habríamos simplificado mucho nuestro código, pero entonces no hubiéramos aprendido cómo funciona internamente la comunicación y transferencia de datos con esta plataforma, conocimiento que nos vendrá bien para comprobar que muchos servicios parecidos a Cosm.com (listados en el párrafo siguiente) trabajan de una forma similar. Además de la librería oficial, también existen otras librerías de terceros que realizan la misma labor igual de bien; en este sentido, podemos probar la "Arduino Cosm Library", disponible en https://github.com/blawson/PachubeArduino o también la librería "PachubeLibrary", disponible en http://code.google.com/p/pachubelibrary.

Cosm.com es muy completo y puede realizar muchísimas funciones más no explicadas aquí. Pero no es el único sitio web de estas características: una alternativa muy interesante a Cosm.com es por ejemplo Nimbits (http://www.nimbits.com), plataforma que nos ofrece una librería oficial para que nuestros sketches Arduino puedan interaccionar con ella fácilmente, y que, además, nos ofrece su código fuente

interno para que podamos montar un servidor Nimbits en nuestro propio computador si así lo deseamos. Otros servicios similares son también Thingspeak (https://www.thingspeak.com), OpenSense (http://open.sen.se) o SensorMonkey (https://sensormonkey.eeng.nuim.ie). En todo caso, recomiendo leer la documentación y guías de cada uno de estos servicios para comprobar cuál es el que se adapta mejor a nuestras necesidades.

Caso práctico: obtención de datos provenientes de Cosm.com

Igual que podemos enviar desde nuestra placa Arduino datos de sensores locales a la web de Cosm.com, nuestra placa Arduino también puede obtener datos de sensores remotos (provenientes de nuestros "feeds" o de los de otros usuarios que los hayan hecho públicos) y utilizar esta información según nos convenga. De esta forma, podríamos controlar una instalación Arduino mediante una entrada de datos provenientes de cualquier rincón del mundo. Por ejemplo, según la temperatura detectada en Hong-Kong podríamos encender un LED conectado a un Arduino situado en Montreal.

Estos datos podemos obtenerlos en tres formatos distintos: CSV, JSON o XML. Por tanto, para poder recuperar la información deseada deberíamos ser capaces de comprender la estructura interna del formato elegido. Afortunadamente, esto no será necesario porque lo que haremos será utilizar una librería que nos facilitará mucho las cosas. Se trata de la librería "PachubeLibrary", descargable de la página http://code.google.com/p/pachubelibrary. Esta librería permite de forma muy sencilla tanto el envío de datos de sensores locales (que es lo que hemos realizado en el caso práctico anterior), como la obtención de datos de sensores remotos (que es para lo que la usaremos). Para ambos casos ofrece códigos de ejemplo, y el mostrado a continuación se basa en uno de ellos. Concretamente, el sketch siguiente obtiene cada cinco segundos los "datastreams" pertenecientes al "feed" y "apikeys" especificados, y los muestra por el canal serie.

```
#include <SPI.h>
#include <Ethernet.h>
#include <ERxPachube.h>
byte mac[] = { 0xCC, 0xAC, 0xBE, 0xEF, 0xFE, 0x91 };
byte ip[] = { 192, 168, 1, 177 };
const char APIKEY[]="La APIKEY de api.cosm.com se ha de poner aquí";
//Aquí se ha de poner nuestro "feed ID"
const char FEEDID[]="00000";
//Creo un objeto que manejará los datos recibidos del FeedID
ERxPachubeDataIn datos(APIKEY,FEEDID);
```



```
void setup() {
       Serial.begin(9600);
       Ethernet.begin(mac, ip);
}
void loop() {
      int estadoConexion;
      word numDatastreams, i;
//Conecto con el FeedID
      estadoConexion = datos.syncPachube();
//Si la conexión es correcta, recibiremos un código HTTP 200 (OK)
       Serial.println(estadoConexion);
//Cuento cuántos datastreams tiene el FeedID (puede haber más de uno)
       numDatastreams = datos.countDatastreams();
       Serial.println("<Sensor>,<Valor>");
       for(i = 0; i < numDatastreams; i++){
              //Obtengo el nombre del sensor del datastream "i"
              Serial.print(datos.getIdByIndex(i));
              Serial.print(",");
              //Obtengo el valor del sensor del datastream "i"
              Serial.print(datos.getValueByIndex(i));
              Serial.println();
       }
      delay(5000);
}
```

Caso práctico: envío de datos a Google Spreadsheets

Si lo que queremos es almacenar los datos leídos por nuestros sensores en una hoja de cálculo online, y así poder realizar diferentes operaciones estadísticas, visualizar gráficos a partir de la información obtenida, descargar esta información en nuestro computador o bien compartirla con otros usuarios (entre muchas otras posibilidades más), una de las maneras más prácticas es utilizar el servicio "Google Spreadsheets", perteneciente a la suite "Google Docs". Para ello, necesitaremos tener una cuenta de usuario Google.

El truco está en crear un "Google Form" que recibirá los diferentes datos enviados por nuestra placa Arduino. "Google Form" es un servicio que permite generar formularios online de forma muy sencilla y rápida, ideal para encuestas o cuestionarios. Lo bueno de los "Google Form" es que automáticamente importan en "Google Spreadsheets" los datos recibidos, sin tener que hacer absolutamente nada.

Para crear un formulario de Google, deberemos dirigirnos a http://docs.google.com y pulsar en el botón “Create->Form”. Deberemos darle un título al formulario y a todas las preguntas que creemos. Las preguntas que añadamos (mediante el botón “Add ítem”) deben ser siempre de tipo TEXT. Los títulos de las preguntas serán los nombres de las columnas de la hoja de cálculo, y cada respuesta obtenida se almacenará en una fila diferente de esa hoja de cálculo. Una vez completado el formulario, deberemos clicar en el botón “Save”. En la parte inferior de la página nos aparecerá un enlace para visualizar el formulario ya acabado. Cliquemos en él para fijarnos en la barra de direcciones del navegador, porque allí veremos la “formkey” del formulario, que es una cadena larga de números y letras que identifica ese formulario entre todos los demás. Esta formkey la deberemos incluir en nuestro sketch Arduino para que nuestra placa pueda interactuar correctamente con él.

También nos debemos fijar en otro detalle importante para nuestro sketch: las cajas de tipo TEXT en el formulario de Google suelen tener un nombre interno identificativo como “entry.0.single” (para la primera pregunta), “entry.1.single” (para la segunda), etc. No obstante, si modificamos la estructura básica del formulario estos nombres pueden cambiar; para asegurarnos de que no lo hayan hecho, es buena idea consultar el código HTML del formulario ya acabado y presentable. Si usamos el navegador libre y multiplataforma Firefox, esto se puede hacer simplemente pulsando a la vez la combinación de teclas CTRL y U.

Ya tenemos creado el formulario, y por tanto, ya es capaz de almacenar en “Google Spreadsheets” los datos introducidos de forma interactiva. Pero si queremos que sea nuestra placa Arduino quien los introduzca de forma autónoma, debemos saber que esta introducción se puede realizar mediante el envío a Google de la dirección:*https://spreadsheets.google.com/formResponse?formkey=MIFORMKEY&ifq&NOMBREINTERNOPREGUNTA1=VALOR&NOMBREINTERNOPREGUNTA2=VALOR&submit=Submit*

En el código siguiente se muestra cómo guardar en un Google Spreadsheet cada dos segundos aproximadamente los valores de dos sensores analógicos conectados a los pines nº 0 y nº 1. Si no hubiera ningún sensor conectado en esos pines, se obtendrán valores aleatorios provenientes del ruido ambiental.

```
#include <SPI.h>
#include <Ethernet.h>
//Se ha de sustituir por la “formkey” de nuestro Google Form
char formkey[] = "DdMBUd3xmTQ52Yvx2XZ01V83VUp2U06EQM";
```



```
byte mac[] = {0x90,0xA2,0xDA,0x00,0x55,0x8D};
byte ip[] = {192,168,1,177};
byte subnet[] = {255,255,255,0};
byte servdns[] = { 8,8,8,8};
byte gateway[] = {192,168,1,1};
char miservidor[] = "spreadsheets.google.com";
String datos="";
EthernetClient micliente;
void setup(){
    Ethernet.begin(mac, ip , servdns, gateway , subnet);
    //Me espero para darle tiempo al shield Ethernet a inicializarse
    delay(1000);
}
void loop(){
        //Creo la cadena a enviar, concatenando sus diferentes partes
         datos="entry.0.single=";
         datos=datos + analogRead(0);
         datos=datos + "&entry.1.single=";
         datos=datos + analogRead(1);
         datos=datos + "&submit=Submit";
         if (micliente.connect(miservidor,80)!=0) {
            micliente.print("POST /formResponse?formkey=");
            micliente.print(formkey);
            micliente.println("&ifq HTTP/1.1");
            micliente.println("Host: spreadsheets.google.com");
            micliente.println("Content-Type:   application/x-www-form-
urlencoded");
            micliente.println("Connection: close");
            micliente.print("Content-Length: ");
            micliente.println(datos.length());
            micliente.println();
            micliente.print(datos);
            micliente.println();
         }
         delay(1000);
         if (micliente.connected()==true) { micliente.stop(); }
         delay(1000);
}
```

En el sketch anterior, para enviar los datos a Google hemos hecho uso del método POST en vez del método GET visto en anteriores ejemplos. La diferencia entre ambos métodos es que con GET los datos recibidos por el formulario se envían como una parte más de la dirección de la página solicitada (y por tanto, aparecen visibles en la barra de direcciones del navegador) y con POST los datos se envían

como una cabecera más de cliente, situada al final de todas las demás y ubicada entre líneas en blanco.

Caso práctico: envío de notificaciones a Pushingbox.com

Pushingbox.com (http://www.pushingbox.com) es un servicio de notificaciones online. Para poderlo probar necesitamos disponer previamente de una cuenta de usuario de Google y de una placa Arduino provista de sensores preparados para detectar algún evento. La placa deberá ejecutar un sketch específico, el cual enviará una notificación a Pushingbox.com cada vez que ese evento ocurra. Lo interesante está en que Pushingbox.com se encarga de reenviar este mensaje a los (múltiples) destinos que hayamos configurado: podemos emitir la notificación a una cuenta de Twitter, a un buzón de correo electrónico, o a alguna aplicación específica para teléfonos móviles de última generación, entre otros. De esta forma, podemos estar informados en cualquier momento y en cualquier lugar de lo que ocurre con nuestra instalación Arduino.

Los pasos para utilizar este servicio son los siguientes. Primero deberemos iniciar sesión con una cuenta de usuario de Google e ir al apartado "My services". Allí deberemos clicar en "Add a service". Veremos que tenemos varias posibilidades de "receptores" de las notificaciones enviadas por Pushingbox: podemos elegir un determinado buzón de correo, una cuenta de Twitter o diferentes aplicaciones para móviles de última generación (concretamente, Prowl o Pushme.to para teléfonos iPhone, Notifry o "Notify my Android" para teléfonos Android o Toasty para teléfonos Windows Phone). Por cada "receptor" seleccionado (podemos elegir varios a la vez) nos aparecerá un botón "Submit" que deberemos clicar y que nos conducirá a un proceso diferente según el tipo de "receptor" elegido. En el caso de utilizar aplicaciones para móviles, en este paso deberemos introducir la "API Key" concreta que se generó en instalar esa aplicación en nuestro aparato.

Una vez realizado este paso, debemos continuar yendo al apartado "My scenarios". Allí deberemos darle un nombre a cada escenario que queramos crear y clicaremos en "Create a scenario". Crear un escenario nos permite básicamente obtener un "DeviceID", que es una cadena identificativa que deberemos introducir en el sketch ejecutado por Arduino, y que nos permitirá identificarnos en Pushingbox. Una vez creado el escenario (o escenarios) deseado, tendremos que decidir qué "receptores" de los definidos en el paso anterior queremos utilizar. Para ello, deberemos clicar en "Add an action". Dependiendo del "receptor", nos preguntarán diferentes detalles (si es un correo nos dirá qué asunto y qué texto queremos añadir, si es un envío a Twitter nos dirá qué mensaje queremos añadir, etc.). Y ya está.

Disponemos del botón "Test" para probar que el envío de notificaciones para esa acción se realice correctamente, pero lo interesante es que este envío lo realice la placa Arduino.

Para ello, podemos utilizar el sketch de ejemplo mostrado a continuación. Para probarlo deberemos tener un pulsador conectado al pin de entrada digital nº 3 de nuestra placa Arduino. Cada vez que se pulse, el sketch lo notificará a PushingBox (y este lo renotificará a los "receptores" que tengamos configurados para el escenario asociado al DevID especificado en el código). Cada vez que se deje de pulsar también se realizará una notificación. Evidentemente, este ejemplo tan básico es solo una referencia para poder añadir a partir de aquí diferentes sensores más sofisticados.

```
#include <SPI.h>
#include <Ethernet.h>
byte mac[] = { 0x00, 0xAA, 0xBB, 0xCC, 0xDE, 0x19 };
/*En la siguiente línea se ha de poner el DevID correspondiente al
escenario que se desea monitorizar. Se pueden utilizar múltiples
DevIDs en un mismo sketch (asociados a diferentes pines de la placa
Arduino).*/
String DEVID1="Tu_DevID_Aqui";
char miservidor[] = "api.pushingbox.com";
boolean estadopulsador = false;
EthernetClient micliente;
void setup() {
  //Pin de entrada digital donde tengo conectado el pulsador
  pinMode(3, INPUT);
  /*Si hay error, no hago nada más y si está todo ok, continúo.
  Presupongo que hay un servidor DHCP en mi LAN que me ofrece
  todos los datos de conectividad */
  if (Ethernet.begin(mac) == 0) { while(true){;} }
  delay(1000);
}
void loop(){
      //Si apreto el pulsador y antes no estaba apretado
      if (digitalRead(3) == HIGH && estadopulsador == false){
        estadopulsador = true;
        //Envío la notificación correspondiente a Pushingbox
        enviarAPushingBox(DEVID1);
      }
      //Si dejo de apretar el pulsador
       if (digitalRead(3) == LOW && estadopulsador == true){
        estadopulsador = false;
        //Envío la notificación correspondiente a Pushingbox
```

```
        enviarAPushingBox(DEVID1);
      }
}
//Función que envía las notificaciones a Pushingbox
void enviarAPushingBox(String devid){
  if (micliente.connect(miservidor, 80)) {
    micliente.print("GET /pushingbox?devid=");
    micliente.print(devid);
    micliente.println(" HTTP/1.1");
    micliente.print("Host: ");
    micliente.println(miservidor);
    micliente.println("User-Agent: Arduino");
    micliente.println();
  }
  micliente.stop();
}
```

Shields alternativos a Arduino Ethernet

Además del shield oficial, existen otros shields de terceros que también añaden conectividad Ethernet (TCP/IP) a una placa Arduino UNO y que podemos elegir en función de sus características. Shields que incorporan el mismo chip Wiz5100 que el shield oficial son por ejemplo el “Ethernet shield” de DFRobot o el “Ethernet shield” de Seeedstudio (que, sin embargo, no incluyen el zócalo para la tarjeta microSD).

Otros shields incorporan un módulo PoE ya integrado, como el “Ethernet shield with PoE” de Freetronics. Este shield dispone concretamente de unos pines (convenientemente señalados) que permiten adaptar su comportamiento según sea el valor del voltaje de entrada transportado por la señal PoE: si ese voltaje es de hasta 12 V, solo se tendrán que colocar un par de puentes de plástico (“jumpers”) sobre dichos pines y ya está (con esto haremos que el pin PoE “+” del shield se conecte al pin “Vin” de la placa Arduino y el pin "-" se conecte al pin “Gnd”); si ese voltaje es de hasta 28 V, será necesario entonces acoplar a los pines del shield un regulador de voltaje adecuado, tal como el “PR28V” (fabricado también por la propia Freetronics) para no quemar la placa. Este regulador proporciona hasta 1 A de corriente, una tensión de salida de 5 V o 7 V y puede recibir una tensión de entrada de hasta 28 V, por lo que no es compatible con el estándar 802.3af (el cual permite tensiones de hasta 48 V) y por tanto, no está preparado para ser conectado a switches PoE comerciales. Para poder conectar este shield a este tipo de switchs (y poder recibir entonces a través de la señal PoE un voltaje de hasta 48 V), es necesario utilizar otro

regulador más capaz (como el "PoE Regulator 802.3af", proporcionado también por la propia Freetronics).

Otro ejemplo parecido al shield anterior es el "PoEthernet shield" de Sparkfun. Dispone de un par de agujeros marcados como "GND/-" y "Vin/+" para soldar allí pines en los que colocaremos un par de "jumpers" (al igual que en el shield anterior). Estos pines permiten recibir correctamente la señal PoE por el cable Ethernet, señal que puede ser regulada a 5 V y 3,3 V. No se puede aplicar, no obstante, una señal PoE con un voltaje mayor de 12 V. Este shield también incluye un zócalo microSD accesible mediante la librería "SD" oficial.

Si no queremos utilizar un shield completo, sino que deseamos usar un módulo Wiz5100 externo independiente (con conector RJ-45 incorporado), podemos adquirir el "WIZnet W5100 Ethernet/Network Module" de IteadStudio. Este módulo se comunica con el exterior mediante el protocolo SPI, por lo que debe ser programado mediante la librería "SPI" oficial de Arduino. Otra plaquita muy similar a la anterior es la distribuida por Sparkfun con el código de producto 9473.

Por otro lado, existen shields Ethernet que no incluyen el chip Wiz5100, sino otro chip diferente, el ENC28J60 de Microchip (también comunicado con la placa mediante SPI). Este chip permite un control mucho más detallado de la comunicación por red y una gran libertad en el uso de diferentes protocolos, libertad que el Wiz5100 no es capaz de ofrecer. No obstante, requiere unos conocimientos algo mayores para poder ser programado. Técnicamente, esto es debido a que el chip Wiz5100 ofrece una pila de protocolos TCP/IP ya integrada en el hardware, de forma que el usuario no los tiene que programar desde cero; esto ofrece la ventaja de no tener que "mancharse las manos" trabajando a bajo nivel, pero ofrece la desventaja de no ser tan flexible y versátil como el chip ENC28J60.

Ejemplos de shields de que incorporan el chip ENC28J60 son el "Gate 0.5" de Snootlab o el "IE shield" de Iteadstudio (que además incorpora un módulo PoE y un zócalo microSD), los cuales se han de programar con alguna librería diferente de la librería "Ethernet" oficial de Arduino. La librería más utilizada es la llamada EtherCard (http://jeelabs.net/projects/cafe/wiki/EtherCard. Otra menos conocida es la librería disponible en https://github.com/turicas/Ethernet_ENC28J60. También es posible añadir la funcionalidad del chip ENC28J60 a nuestra placa Arduino UNO mediante un módulo SPI (como el "ENC28J60 Ethernet module" de Iteadstudio), sin necesidad por tanto de utilizar un shield al completo.

Comunicación por red usando una placa Arduino UNO estándar

Es posible conectar una placa Arduino UNO a una red Ethernet sin necesidad de utilizar ni el shield Arduino Ethernet ni la placa Ethernet propiamente dicha: solamente con la ayuda de un computador ya es posible. Concretamente, conectando nuestra placa UNO vía USB a un computador con acceso a la red, podremos usar este como intermediario para reenviar los mensajes de la placa UNO al exterior (y en sentido contrario también).

Para ello, necesitamos ejecutar un programa en el computador que establezca por el puerto USB una comunicación serie con la placa Arduino (para recibir datos de ella –de sensores, normalmente– y también para enviarle señales de control –a actuadores, normalmente–) a la vez que establezca por el puerto Ethernet una comunicación de red con el resto de la LAN e Internet. Es decir, que actúe como un "puente" entre la red y una placa Arduino sin capacidad para conectarse a ella por sí misma. A estos programas se les llama genéricamente "proxies serie-red".

Un "proxy serie-red" normalmente lo deberemos desarrollar nosotros mismos, ya que su funcionalidad dependerá mucho del diseño de nuestro proyecto y del tipo de mensajes que queramos transmitir o recibir del exterior. Por tanto, debemos conocer algún lenguaje de programación que nos permita crear ese programa intermediario capaz a la vez de contactar con Arduino vía puerto serie y el resto del mundo vía Ethernet.

Se escapa por completo de los objetivos de este libro profundizar en el uso de otros lenguajes de programación diferentes del propio de Arduino, pero a modo de referencia, podemos nombrar lenguajes que han sido suficientemente probados en estas circunstancias. Uno de ellos es naturalmente Processing, lenguaje muy parecido a Arduino enfocado en el desarrollo de aplicaciones visuales y animadas. Incorpora una librería llamada "Serial" (http://processing.org/reference/libraries/serial) y otra librería llamada "Network" (http://processing.org/reference/libraries/net), que permiten vincular en un mismo programa los dos extremos de la comunicación (serie y Ethernet).

Otro lenguaje diferente de Processing, pero también libre y multiplataforma es Python (http://www.python.org). Este lenguaje es multipropósito, ya que puede ser utilizado para el desarrollo de aplicaciones de tipo binario ("ejecutable") o bien de tipo web. En el caso que nos ocupa, para la comunicación vía Ethernet existen varias posibilidades, pero la más versátil y potente es usar la llamada "Socket", integrada dentro del intérprete del lenguaje por defecto. Para la comunicación vía serie en

cambio no disponemos de ninguna librería por defecto, por lo que deberíamos de descargar e instalar una de terceros. La más recomendable es la librería "PySerial", disponible en http://pyserial.sourceforge.net.

Otro lenguaje, libre y multiplataforma es PHP (http://www.php.net), lenguaje multipropósito que puede ser utilizado también para el desarrollo de aplicaciones binarias o (sobre todo) de tipo web. Ofrece una sintaxis mucho más parecida al lenguaje Arduino que Python, debido a que, al igual que aquel, está basado en C. En el caso que nos ocupa, para la comunicación vía Ethernet existen varias posibilidades, pero la más versátil y potente es usar la llamada "Sockets", integrada dentro del intérprete del lenguaje por defecto. Para la comunicación vía serie en cambio, no disponemos (al igual que en Python) de ninguna librería por defecto, por lo que deberíamos de descargar e instalar una de terceros. La más recomendable es la librería "PhpSerial", disponible en http://code.google.com/p/php-serial.

Por otro lado, algo muy habitual cuando se utiliza la configuración de computador intermediario que estamos describiendo es recopilar datos procedentes de Arduino para almacenarlos en un servidor de bases de datos remoto. En ese caso, más que una librería Ethernet genérica, el lenguaje de programación con el que esté desarrollado el proxy serie-red ha de proporcionar librerías específicas para la conexión y trabajo con servidores específicos de bases de datos. Todos los lenguajes mencionados (Processing, Python o Php) disponen de esta funcionalidad.

Si no tenemos los conocimientos (o el tiempo) suficientes para desarrollar nuestro propio proxy serie-red genérico, también podemos elegir alguno ya desarrollado y listo para ser instalado y utilizado al momento. Así no nos tendremos que preocupar de desarrollar nada "a mano" y tan solo deberemos ocuparnos de aprender a usar ese software en concreto; lo malo es que puede que dicho software no se adapte a nuestras necesidades particulares. La idea es en todos los casos la misma: los sketches de nuestra placa Arduino se comunicarán con nuestro computador utilizando el objeto Serial y nuestro computador, mediante el proxy que hayamos instalado, se encargará a abrir un puerto TCP/IP (configurable) para hacerlo disponible al resto del mundo.

Un ejemplo de proxy serie-red (solo funcional en sistemas Windows, no obstante) es Bloom, disponible en https://sensormonkey.eeng.nuim.ie. Un proxy serie-red que funciona en sistemas Linux es Ser2net (http://ser2net.sourceforge.net). Un proxy serie-red que funciona en ambos sistemas es SerProxy, disponible en http://www.lspace.nildram.co.uk/freeware.html.

COMUNICACIÓN A TRAVÉS DE WI-FI

¿Qué es Wi-Fi?

El funcionamiento aparente de una red Wi-Fi es muy similar al de una red Ethernet, solo que sin cables. No obstante, además de las direcciones IP y las direcciones MAC, en esta tecnología inalámbrica hay que tener en cuenta otros conceptos:

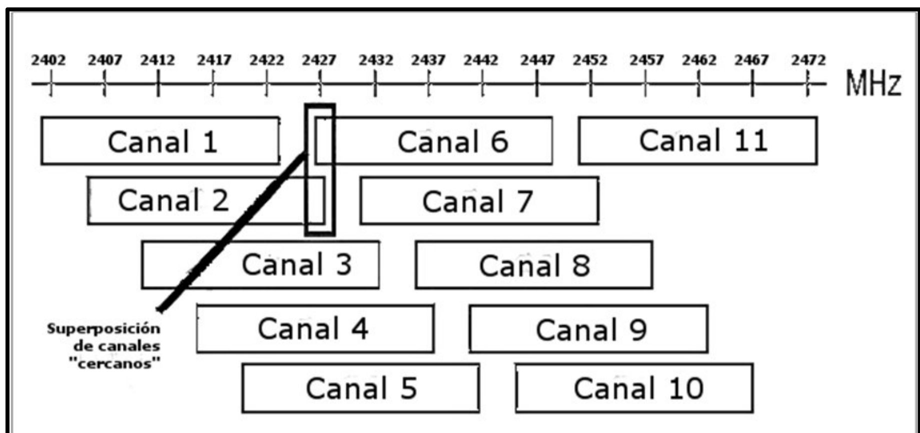
Estándar IEEE802.11: "Wi-Fi" se basa en este estándar, el cual es en realidad un conjunto de estándares. Dependiendo de la compatibilidad con uno o más de dichos estándares, encontraremos dispositivos que pueden formar parte de redes Wi-Fi 802.11b, 802.11g o 802.11n entre otros. La diferencia más importante entre ellos es la velocidad de transmisión de datos (de hasta 11 Mbits/s, 54 Mbits/s y 600 Mbits/s, respectivamente), pero las tres especificaciones operan en la banda de los 2,4 GHz (el 802.11n además en la de 5,4 GHz) y su rango de señal llega hasta el centenar de metros.

Punto de acceso (AP): un punto de acceso es un equipo de red inalámbrico (puede ser un computador con el software adecuado, o un dispositivo hardware específico) que se encarga de gestionar de forma centralizada las comunicaciones de todos los dispositivos que forman la red Wi-Fi. No solo se utiliza para controlar las comunicaciones internas de la red, sino que también hace de puente en las comunicaciones con las redes externas (redes Ethernet e Internet), a modo de "transformador de señal" entre redes inalámbricas y cableadas. Pueden existir redes Wi-Fi que no dispongan de punto de acceso; en este caso, las comunicaciones se llevan a cabo directamente entre los distintos terminales que forman la red y su estructura se suele denominar "ad-hoc", (o también de igual a igual" –"peer-to-peer"– ó IBSS). Las redes Wi-Fi que disponen de punto de acceso tienen una estructura llamada "en infraestructura" (o "BSS").

Modo: un dispositivo Wi-Fi puede tener un rol determinado dentro de la red, y esto se configura estableciendo su modo de funcionamiento. El modo Station o ("Managed") es el modo en el que un dispositivo es un mero cliente que se conecta a un punto de acceso para tener conectividad. El modo AP o ("Master") es el modo en el que un dispositivo puede trabajar él mismo como punto de acceso (si dispone del firmware adecuado). Existen otros modos más exóticos que no veremos.

SSID: es un dato emitido por el punto de acceso que identifica la red inalámbrica a la que pertenece. En otras palabras, es el "nombre de la red" que los terminales son capaces de ver para poderse conectar. El SSID se puede modificar accediendo a la configuración del punto de acceso. No confundir con el BSSID, que representa la dirección MAC del punto de acceso.

Canal: la banda de frecuencias electromagnéticas en la que trabaja una red Wi-fi (la banda de los 2,4 GHz, generalmente) se divide en varios canales. Concretamente, el estándar subdivide el rango de los 2,4 GHz en 14 canales separados entre sí por 5 MHz (aunque cada país aplica sus propias restricciones al número de canales disponibles: en Europa solo se pueden utilizar del canal 1 al 13, por ejemplo). El problema de esta distribución es que cada canal necesita 22 MHz de ancho de banda para operar, y como se puede apreciar en la figura siguiente, esto produce un solapamiento de varios canales contiguos (por ejemplo, el canal 1 se superpone con los canales 2, 3, 4 y 5). La consecuencia de que los dispositivos emitan en rangos solapados es que generan interferencias entre sí, dificultando la conectividad y velocidad de la red. Así pues, en la medida de lo posible, se deberá elegir canales diferentes para evitar esto (lo más habitual es elegir los canales 1, 6 y 11).



Algoritmo de cifrado: la comunicación entre los dispositivos que forman una red Wi-Fi se puede cifrar, de tal forma que los datos transmitidos entre ellos no puedan ser conocidos por dispositivos ajenos a ella. Así se dota de seguridad y confidencialidad a unas comunicaciones que son fácilmente captadas de por sí (ya que el medio es el aire). Existen diferentes métodos de cifrado: el sistema WEP (no recomendable debido a su debilidad ante ataques de descifrado), el WPA-TKIP (más seguro que el WEP pero no lo suficiente) o

el más actual WPA2-CCMP (también conocido como WPA2-AES o 802.11i, implementa una mayor protección de datos y un mejor control de acceso), entre otros. Dentro del método WPA2-CCMP podemos elegir varias tecnologías de almacenamiento de contraseñas: la PSK (llamada también "WPA2 Personal") o la EAP (llamada también "WPA2 Enterprise"); el WPA2-Personal es más adecuado para instalaciones inalámbricas domésticas debido a su rápida y sencilla configuración, y el WPA2-Enterprise está pensado para instalaciones corporativas donde se requiere la seguridad adicional ofrecida por un servidor central de contraseñas (de tipo RADIUS). En cualquier caso, todos los métodos de cifrado estudiados en este libro requerirán la introducción dentro de la configuración del punto de acceso de una contraseña (o "clave de paso"), que deberá ser conocida por los dispositivos que se quieran unir a él.

Uso del Arduino WiFi Shield y de la librería oficial WiFi

Si para utilizar comunicación Wi-Fi elegimos trabajar con el shield oficial de Arduino (el "Arduino WiFi Shield"), deberemos programar nuestros sketches con la ayuda de la librería "WiFi". Esta librería es bastante similar a la librería "Ethernet" vista en apartados anteriores, por lo que tan solo se mencionarán las diferencias más importantes y se remitirá al estudio de la librería "Ethernet" para conocer los detalles.

Una vez importada la librería, lo primero que deberemos hacer es inicializarla mediante la función *WiFi.begin()*, la cual es en cierta medida equivalente a la ya conocida *Ethernet.begin()*. Esta función puede ser escrita de tres maneras diferentes, devolviendo en todas ellas el valor constante predefinido WL_CONNECTED si consigue conectar con la red Wi-Fi especificada o el valor WL_IDLE_STATUS si no.

WiFi.begin(ssid): donde "ssid" representa el SSID de la red Wi-Fi a la que nos queremos conectar. Esta forma de escribir *WiFi.begin()* es la que debemos usar cuando la red Wi-Fi es de tipo abierto (sin cifrar).

WiFi.begin(ssid, pass): donde "pass" representa la contraseña de la red Wi-Fi protegida mediante WPA2-Personal a la que nos queremos conectar.

WiFi.begin(ssid, indice, key): si nos queremos conectar a una red Wi-Fi protegida mediante WEP, debemos proporcionar dos datos: "índice" y "key". Las contraseñas WEP son cadenas de 10 o 26 caracteres hexadecimales (parámetro "key") y como pueden existir hasta cuatro diferentes en una

misma red, el parámetro “índice” sirve para de indicar cuál de ellas (0, 1, 2 o 3) se quiere utilizar.

Como se puede comprobar, en *WiFi.begin()* no se puede especificar ninguna ip fija (ni máscara, ni puerta de enlace ni servidores dns), ya que se presupone que toda esta información la asigna dinámicamente mediante DHCP el punto de acceso al que se conecte el shield en ese momento.

Otras funciones generales interesantes de la librería WiFi son:

WiFi.disconnect(): desconecta el shield de la red actual. No tiene parámetros ni valor de retorno.

WiFi.scanNetworks(): escanea las redes Wi-Fi disponibles y devuelve el número total de redes encontradas –dato de tipo byte–. No tiene parámetros.

WiFi.SSID(): su valor de retorno es una cadena que contiene el SSID de la red a la que está conectado actualmente el shield. Opcionalmente, se puede indicar un parámetro –de tipo byte– que representa la posición numérica (0,1,2…) dentro de la lista de redes encontradas por *WiFi.scanNetworks()* que tiene la red de la cual se quiere conocer su SSID (aunque no sea la red a la que estamos conectados actualmente).

WiFi.BSSID(): no tiene valor de retorno, pero sí un parámetro obligatorio que ha de ser de tipo array de 6 posiciones de tipo byte. Allí se guardará la dirección MAC del punto de acceso al cual el shield está conectado en este momento. El elemento 0 del array corresponde al byte de la MAC de más a la derecha, y el 5 el de más a la izquierda.

WiFi.RSSI(): devuelve un valor –de tipo long– que representa la fuerza de la señal recibida en la conexión actual (el llamado “Received Signal Strength”, medido en dBm). Opcionalmente, se puede indicar un parámetro –de tipo byte– que representa la posición numérica (0, 1, 2…) dentro de la lista de redes encontradas por *WiFi.scanNetworks()* que tiene la red de la cual se quiere conocer este dato (aunque no sea la red a la que estamos conectados actualmente).

WiFi.encryptionType(): devuelve un valor –de tipo byte– que representa el tipo de encriptación de la red actual. Un 2 equivale a WPA-TKIP, un 4 a WPA2-

CCMP, un 5 a WEP y un 7 a red abierta. Opcionalmente, se puede indicar un parámetro –de tipo byte– que representa la posición numérica (0, 1, 2…) dentro de la lista de redes encontradas por *WiFi.scanNetworks()* que tiene la red de la cual se quiere conocer este dato (aunque no sea la red a la que estamos conectados actualmente).

WiFi.macAdress(): no tiene valor de retorno, pero sí un parámetro obligatorio que ha de ser de tipo array de 6 posiciones de tipo byte. Allí se guardará la dirección MAC del propio shield. El elemento 0 del array corresponde al byte de la MAC de más a la derecha, y el 5 el de más a la izquierda.

WiFi.localIP(): devuelve la dirección ip actual del shield. Este valor de retorno ha de ser declarado previamente de tipo "IPAddress" (el cual no deja de ser un simple array de 4 posiciones numéricas). Esta función no tiene parámetros. Sería equivalente a la función *Ethernet.localIP()*.

WiFi.subnetMask(): devuelve la máscara de red actual del shield. Este valor de retorno ha de ser declarado previamente de tipo "IPAddress". Esta función no tiene parámetros.

WiFi.gatewayIP(): devuelve la dirección ip de la puerta de enlace actualmente configurada en el shield. Este valor de retorno ha de ser declarado previamente de tipo "IPAddress". Esta función no tiene parámetros.

Tal como ocurre con la librería Ethernet, en la librería WiFi disponemos tanto de objetos de tipo servidor como objetos de tipo cliente, y en ambas librerías su funcionamiento y comportamiento es exactamente igual:

Para crear un objeto servidor: debemos declararlo mediante una línea parecida a esta: `WiFiServer miservidor(80);`, donde "miservidor" es el nombre que hemos elegido para el objeto de tipo WiFiServer, y el número entre los paréntesis representa el puerto que se abre a la espera de las conexiones entrantes de los clientes. A partir de aquí, podremos utilizar las funciones ya conocidas **miservidor.begin()**, **miservidor.available()** –la cual, en este caso, devuelve un objeto de tipo "WiFiClient"–, **miservidor.write()**, **miservidor.print()** y **miservidor.println()**.

Para crear un objeto cliente: debemos declararlo mediante una línea parecida a esta: `WiFiClient micliente;` donde "micliente" es el nombre

que hemos elegido para el objeto de tipo WiFiClient. A partir de aquí, podremos utilizar las funciones ya conocidas **micliente.connected()**, **micliente.connect()** –donde el servidor se puede especificar por su ip o por su nombre DNS–, **micliente.write()**, **micliente.print()**, **micliente.println()**, **micliente.available()**, **micliente.read()**, **micliente.flush()** y **micliente.stop()**.

A continuación, mostraremos unos cuantos sketches que inciden en las novedades que aporta esta librería respecto a la librería Ethernet. Los ejemplos presentados anteriormente en el apartado correspondiente a la librería Ethernet continúan siendo perfectamente válidos con la librería WiFi: tan solo se ha de cambiar el modo de establecimiento de la conexión con *WiFi.begin()* y el tipo de los objetos a usar (WiFiServer y WiFiClient).

Ejemplo 8.4: El siguiente código no se conecta a ninguna red Wi-Fi: lo que hace es mostrar por el canal serie las redes detectadas (y de regalo, la dirección MAC del shield). Hay que aclarar que posiblemente, el shield no detecte tantas redes como un computador, ya que estos suelen incorporar una antena de mayores dimensiones.

```
#include <SPI.h>
#include <WiFi.h>
void setup() {
  Serial.begin(9600);
  printMacAddress(); //Muestra la dirección MAC del shield
}
void loop() {
  listNetworks(); //Muestra información de las redes WiFi disponibles
  delay(10000);
}
void printMacAddress() {
  byte mac[6];
  WiFi.macAddress(mac);
  Serial.print("MAC: ");
  Serial.print(mac[5],HEX);  Serial.print(":");
  Serial.print(mac[4],HEX);  Serial.print(":");
  Serial.print(mac[3],HEX);  Serial.print(":");
  Serial.print(mac[2],HEX);  Serial.print(":");
  Serial.print(mac[1],HEX);  Serial.print(":");
  Serial.println(mac[0],HEX);
}
void listNetworks() {
  byte numSsid = WiFi.scanNetworks();
  Serial.print("Numero de redes disponibles: ");
```

```
  Serial.println(numSsid);
  /*Muestro el índice de cada red encontrada, además de su SSID,
  su fuerza de señal y su cifrado utilizado */
  for (int i = 0; i<numSsid; i++) {
   Serial.print(i); Serial.print(") "); Serial.print(WiFi.SSID(i));
   Serial.print("\tSeñal (en dBm): ");  Serial.print(WiFi.RSSI(i));
   Serial.print("\tCifrado:");Serial.println(WiFi.encryptionType(i));
  }
}
```

Ejemplo 8.5: El código siguiente muestra cómo conectar el shield a una red WiFi abierta, de nombre “mired”.

```
#include <SPI.h>
#include <WiFi.h>
char ssid[] = "mired";
int status = WL_IDLE_STATUS;
void setup() {
  Serial.begin(9600);
  status = WiFi.begin(ssid);
  if (status != WL_CONNECTED) {
    Serial.println("Error en la conexión");
  } else {
    Serial.println("Conectado");
  }
}
void loop() {}
```

Ejemplo 8.6: El código siguiente muestra cómo conectar el shield a una red WiFi de nombre “mired” y cifrada mediante WPA2-Personal con la contraseña “12345678”.

```
#include <SPI.h>
#include <WiFi.h>
char ssid[] = "mired";
char pass[] = "12345678";
int status = WL_IDLE_STATUS;
void setup() {
  Serial.begin(9600);
  status = WiFi.begin(ssid, pass);
  if (status != WL_CONNECTED) {
    Serial.println("Error en la conexión ");
  } else {
```

```
    Serial.println("Conectado");
  }
}
void loop() {}
```

Ejemplo 8.7: El código siguiente muestra cómo conectar el shield a una red WiFi de nombre “mired” y cifrada mediante WEP con una clave de 10 caracteres hexadecimales e índice 0.

```
#include <SPI.h>
#include <WiFi.h>
char ssid[] = "mired";
char key[] = "ABBADEAF01";   //La clave de 10 caracteres
int keyIndex = 0;            //El índice de la clave dentro del AP
int status = WL_IDLE_STATUS;
void setup() {
  Serial.begin(9600);
  status = WiFi.begin(ssid, keyIndex, key);
  if (status != WL_CONNECTED) {
    Serial.println("Error en la conexión ");
  } else {
    Serial.println("Conectado");
  }
}
void loop() {}
```

Ejemplo 8.8: Una vez conectado a una red WiFi (del tipo que sea), el shield adquirirá un conjunto de parámetros de red (ip, máscara, puerta de enlace, etc.) gracias al punto de acceso al cual se conecte. En nuestros sketches podemos utilizar la función siguiente para conocer en todo momento el valor de dichos parámetros.

```
void printWifiData() {
  IPAddress ip = WiFi.localIP();
  Serial.print("Dirección IP: ");
  Serial.println(ip);
  IPAddress subnet = WiFi.subnetMask();
  Serial.print("Máscara de red: ");
  Serial.println(subnet);
  IPAddress gateway = WiFi.gatewayIP();
  Serial.print("Puerta de enlace: ");
  Serial.println(gateway);
}
```


Ejemplo 8.9: Una vez conectados también podemos conocer algunos datos de la red a la que nos hemos incorporado (concretamente el SSID de esa red, la fuerza de señal al punto de acceso, el algoritmo de cifrado usado y la MAC de ese punto de acceso) mediante la ejecución en nuestro sketch de la siguiente función:

```
void printCurrentNet() {
  Serial.print("SSID: ");
  Serial.println(WiFi.SSID());
  Serial.print("Fuerza de señal (RSSI):");
  long rssi = WiFi.RSSI();
  Serial.println(rssi);
  Serial.print("Tipo de cifrado:");
  byte encryption = WiFi.encryptionType();
  Serial.println(encryption,HEX);
  //Muestro la MAC del punto de acceso al cual estoy conectado
  byte bssid[6];
  WiFi.BSSID(bssid);
  Serial.print("BSSID: ");
  Serial.print(bssid[5],HEX);
  Serial.print(":");
  Serial.print(bssid[4],HEX);
  Serial.print(":");
  Serial.print(bssid[3],HEX);
  Serial.print(":");
  Serial.print(bssid[2],HEX);
  Serial.print(":");
  Serial.print(bssid[1],HEX);
  Serial.print(":");
  Serial.println(bssid[0],HEX);
}
```

Otros shields y módulos que añaden conectividad Wi-Fi

Además del shield oficial, existen otros shields que también añaden conectividad Wi-Fi a una placa Arduino. En esta categoría podemos encontrar por ejemplo el "Wifly Shield" de Sparkfun (producto nº 9954), que permite conectarse a redes Wi-Fi de tipo 802.11b/g usando varios sistemas de encriptación gracias al módulo Wi-Fi RN-131C que incorpora (del fabricante Roving Network). Al igual que el shield oficial, se comunica con la placa a través de los pines SPI, pero a diferencia de aquel, el "Wifly Shield" funciona a 3,3 V regulados. Otro detalle de este shield es que dispone de una pequeña área de prototipado. La manera más sencilla (aunque no la única) de configurar y gestionar este shield es utilizando la librería disponible en https://github.com/sparkfun/WiFly-Shield.

Otro shield interesante es el "Wifi shield" de DFRobot, el cual incorpora el módulo Wi-Fi WizFi210 de Wiznet. Este módulo aporta conectividad 802.11b/g/n con posibilidad de usar también varios sistemas de encriptación. Además, este shield permite el acoplamiento de una antena externa gracias a su conector de tipo "U.FL". La configuración de este shield es algo diferente a los anteriores: para establecer los datos de conexión del shield (SSID con el que asociarse, la posible contraseña, la indicación de si se solicitará ip vía DHCP o no, etc.) se ha de utilizar un programa de tipo "terminal serie" y enviar una serie de comandos específicos. Una vez hecho esto, el shield ya estará configurado de forma predefinida para conectarse a una red predeterminada. A partir de allí ya podremos recibir y enviar datos a través de ella utilizando en nuestro sketch simplemente las funciones *Serial.print()* y *Serial.read()*. Los usuarios de Windows pueden utilizar el programa gráfico WizSmartScript (descargable de la página oficial de Wiznet) que ayuda a tener configurado este shield en unos pocos pasos sin necesidad de conocer ningún comando concreto. Para más detalles sobre cómo configurar y utilizar este shield, remito a la información disponible en la web del producto.

Otro ejemplo es el shield "Arduino Wifi Shield", de Open-Electronics, el cual incorpora una antena integrada y el chip ZG2100 de Microchip (todo ello encapsulado en forma de módulo, el MRF24WB0MA). Este shield permite utilizar los modos 802.11b/g/n, incluye también un zócalo microSD, funciona a 3,3 V, y se comunica con la placa mediante SPI. Para poder configurarla y utilizarla se puede utilizar una librería específica disponible en http://code.google.com/p/wifi-shield-oe.

Otro shield que incorpora el mismo módulo wi-fi de Microchip es el "CuHead Wifi Shield" de Linksprite, el cual utiliza también el protocolo SPI para comunicarse con la placa, pero ha de ser programado mediante una librería propia, descargable desde https://github.com/linksprite/ZG2100BasedWiFiShield. Linksprite también construye y vende otros shields wi-fi, como el llamado "Anaconda" (el cual puede ser controlado simplemente mediante *Serial.print()* y *Serial.read()*) o el llamado "Juniper" (con el módulo wi-fi GS1011 del fabricante Gainspan). En todo caso, remito a la consulta de las páginas web respectivas para conocer los detalles técnicos de configuración y uso de cada uno de estos shields.

Otro shield similar es el "Arduino Wifi Shield" de Olimex, el cual incorpora el zócalo para incluir un módulo Wiz610wi de Wiznet (el módulo propiamente y la antena se ha de adquirir aparte). Este módulo es capaz utilizar los modos 802.11b/g y se puede programar con una librería específica, descargable desde la propia web de Olimex. Allí también podemos acceder a su guía de instalación, configuración y uso, además de descargar varios códigos de ejemplo.

Otro shield que permite la conectividad 802.11b (y varios métodos de cifrado) es el llamado “Hydrogen” de DIY Sandbox. Contiene un zócalo microSD y se puede programar mediante la librería “Wirefree”, disponible en la página https://github.com/diysandbox/Wirefree. El mismo fabricante ofrece también una placa llamada “Platinum” que es compatible con la placa Arduino UNO y que tiene la capacidad de conectarse vía Wi-Fi sin necesidad de ningún shield (de hecho, ofrece las mismas prestaciones que el shield “Hydrogen” –mismos métodos de cifrado, mismo zócalo microSD, misma librería de programación, etc.– pero en forma de una única placa). Esta placa se programa vía FTDI.

Otro shield es el “RedFly” de Watterott. Este shield permite conectarse a redes de tipo 802.11b/g/n/i con los mismos métodos de cifrado que el resto de shields ya comentados. No dispone de zócalo microSD. Se programa mediante la librería disponible en https://github.com/watterott/RedFly-Shield.

Si hablamos de módulos, Sparkfun comercializa con el código de producto 10050 un módulo llamado “Wifly GSX Breakout” que incluye el mismo chip que viene en su “Wifly Shield”, el RN-131C. Esta plaquita breakout en su configuración más básica tan solo requiere cuatro conexiones: alimentación (3,3 V), tierra, RX (a un pin TX por software de la placa Arduino, para no interferir con la comunicación serie por hardware) y TX (al pin RX de Arduino), y se puede programar mediante la librería “Wifly Serial”, disponible en http://sourceforge.net/projects/arduinowifly.

Otros módulos Wi-Fi (más baratos que el anterior) son los llamados RN-XV, distribuidos también en Sparkfun con códigos de producto 11047, 11048 y 10822. Estos productos se diferencian entre sí en el tipo de conector que proporcionan para la antena: el primero ofrece un conector de tipo “SMA” para enchufarle una antena compatible (como podría ser por ejemplo el producto nº 145 o nº 558), el segundo proporciona un conector de tipo “U.FL” para enchufarle una antena compatible (como podría ser por ejemplo el producto nº 11320) y el tercero proporciona directamente una antena de tipo “wire”. Salvando estas diferencias, todas estas plaquitas breakout permiten conectarse a redes de tipo 802.11b/g y, en su configuración más básica, requieren las mismas cuatro conexiones: alimentación (3,3 V), tierra, RX (a un pin TX por software de la placa Arduino, para no interferir con la comunicación serie por hardware) y TX (al pin RX de Arduino). Todas se pueden programar mediante la misma librería usada con el “Wifly Shield” de Sparkfun (aunque incorporen otro chip, el RN-171).

La ventaja de los módulos RN-XV es que tiene la misma forma y disposición de pines que los módulos XBee, por lo que, de hecho, se podrían utilizar acoplados a la "Arduino Wireless Shield" oficial. Esta característica hace que sean ideales cuando tengamos una instalación ya montada de módulos XBee y deseemos realizar un cambio a la tecnología Wi-Fi de la forma más rápida y sencilla posible: simplemente sustituyendo un módulo por el otro (y modificando los sketches pertinentes, claro) ya debería ser suficiente.

COMUNICACIÓN A TRAVÉS DE BLUETOOTH

¿Qué es Bluetooth?

Bluetooth (http://www.bluetooth.org) es el nombre de una especificación industrial (estandarizada oficialmente con el nombre de IEEE 802.15.1) que define las características de un tipo de redes inalámbricas de corto alcance. Su principal uso es proporcionar un protocolo de comunicación entre distintos dispositivos electrónicos de consumo (computadores, impresoras, teléfonos, cámaras digitales, dispositivos de audio, etc.) relativamente próximos (a unos pocos metros de distancia) sin que haya la necesidad de llevar un control explícito por parte del usuario de direccionamientos de red, permisos y otros aspectos típicos de redes tradicionales. La principal ventaja de usar Bluetooth es que permite simplificar el descubrimiento y configuración automática de dispositivos cercanos, ya que estos pueden indicarse entre sí los servicios que ofrecen de forma autónoma.

El estándar Bluetooth utiliza para la transmisión de voz y datos un enlace de radiofrecuencia en la banda ISM de los 2,4 GHz. Las bandas ISM ("Industrial, Scientific and Medical") están reservadas internacionalmente para el uso no comercial de radiofrecuencia electromagnética. Esto quiere decir que se pueden utilizar abiertamente por todo el mundo sin necesidad de licencia, simplemente respetando las regulaciones que limitan los niveles de potencia transmitida. Existen otras tecnologías inalámbricas que utilizan las mismas bandas ISM, como por ejemplo el estándar inalámbrico "Wi-Fi" (basado en la especificación IEEE802.11, http://www.ieee802.org/11) que precisamente utiliza también la banda de los 2,4 GHz, tal como ya sabemos. No obstante, Bluetooth y Wi-Fi cubren necesidades diferentes: Wi-Fi utiliza una potencia de salida de señal mayor, por lo que lleva a conexiones más sólidas, rápidas, seguras y de mucho mayor alcance, pero requiere una cierta configuración previa (similar a una red Ethernet tradicional).

Dentro del estándar Bluetooth están definidos los llamados “perfiles de dispositivo”. Cada “perfil” es un protocolo adicional superpuesto al protocolo Bluetooth básico que determina la manera de comunicarse que tendrá el dispositivo que lo implemente. Es decir: dependiendo del perfil utilizado, un dispositivo Bluetooth será capaz de recibir/transmitir en un formato de datos concreto y no en otro. Esto implica que para que dos dispositivos Bluetooth se puedan comunicar entre sí, han de utilizar el mismo perfil: no basta con que “sean solo Bluetooth”.

Un dispositivo puede implementar uno o más perfiles, y actuar por tanto de una manera u otra según el uso que se le dé. Los perfiles más habituales dentro del ecosistema Arduino son dos: el SPP (Serial Port Profile) y el HID (Human Interface Device Profile). El primero sirve para que el dispositivo que lo implemente pueda establecer una comunicación de tipo serie con otros dispositivos (al estilo de los chips UART); este es el perfil que tienen los módulos y shields Bluetooth estudiados en este apartado. El segundo sirve para que el dispositivo que lo implemente pueda comportarse como un ratón, un teclado, un joystick o un pulsador (entre otros dispositivos de baja latencia y consumo). De hecho, el perfil HID de Bluetooth es muy parecido al protocolo HID definido en el estándar USB (el cual implementan las placas Leonardo y Due mediante las librerías “Mouse” y “Keyboard”).

Módulos que añaden conectividad Bluetooth

Si queremos dotar a nuestra placa Arduino UNO de la capacidad de comunicarse vía Bluetooth, deberemos conectarle algún módulo receptor/transmisor Bluetooth. Estos módulos funcionan por lo general como dispositivos esclavos, por lo que deberá ser el otro extremo de la comunicación (normalmente un computador o un teléfono móvil con capacidad Bluetooth) el que funcione como dispositivo maestro. Es decir, siempre será el computador/móvil quien inicie la conexión con la placa Arduino y no al revés. Si nuestro computador no dispusiera de ningún emisor/receptor de Bluetooth integrado, podríamos acoplarle uno externo, como por ejemplo el producto nº 9434 de Sparkfun.

Sparkfun distribuye varios módulos, llamados genéricamente “BlueSMiRF”. Dos de ellos (el producto nº 158 y el nº 10268) incorporan el chip RN-41, el cual permite conexiones de hasta distancias de 100 metros. La diferencia entre estos dos módulos consiste en que el primero incluye un conector de tipo “SMA” para enchufar una antena compatible y el segundo incluye la antena ya integrada dentro del PCB del módulo.

Otros módulos "BlueSMiRF" son los productos nº 10269 y nº 10393. Ambos incorporan el chip RN-42 y tienen la antena ya integrada dentro del PCB del módulo. La principal diferencia entre ambos módulos es que el segundo está específicamente diseñado para ser usado junto con las placas Arduino Pro o Arduino Lilypad mediante conexión FTDI. Por otro lado, la principal diferencia entre el chip RN-42 y el RN-41 de los módulos anteriores es que el RN-42 es de "clase 2" mientras que el RN-41 es de "clase 1". Que un dispositivo sea de "clase 2" implica que su distancia máxima útil de detección y transmisión de datos es de tan solo unos 10 metros, pero a cambio, su consumo eléctrico es mucho menor.

En todo caso, todos estos módulos se configuran y utilizan de forma similar. Básicamente lo que hacen es transformar la señal Bluetooth recibida desde el exterior en una señal serie RX para entregarla al ATmega328P de la placa Arduino, y transformar la señal TX generada por el ATmega328P en una señal Bluetooth lista para enviar al exterior. Por tanto, una vez establecido el cableado pertinente, manejar la transferencia de datos vía Bluetooth es tan sencillo como utilizar las funciones del objeto Serial (concretamente, *Serial.print()* para enviar al exterior y *Serial.read()* para recibir de él). La velocidad de comunicación entre el ATmega328P y los módulos BlueSMiRF es por defecto de 115200 bits/s, por lo que este valor deberá ser el especificado en *Serial.begin().*

Los módulos BlueSMiRF disponen de 6 conectores. Dos de ellos no los utilizaremos para nada: se trata de los etiquetados como "CTS" y "RTS". Los otros conectores son triviales: el conector de alimentación "VCC" ha de ser enchufado a una fuente de 3,3 V (aunque los 5 V los acepta también), el conector "GND" a tierra, el conector "TX" ha de ser enchufado al pin-hembra "RX" de la placa Arduino y el conector "RX" ha de ser enchufado al pin-hembra "TX" de la placa. Además, puede existir un séptimo conector algo distanciado de los anteriores etiquetado como "PI04", utilizado para resetear los valores de fábrica del módulo (este tampoco lo utilizaremos). No obstante, para no tener problemas a la hora de configurar estos módulos, no debemos cablear "de golpe" los cuatro conectores (VCC, GND, TX y RX), sino que debemos seguir estos pasos:

1. Cargar en la placa Arduino de la forma habitual (vía USB) el código que queramos que ejecute.

2. Alimentar el módulo enchufando solamente el conector "VCC" y "GND" y situarlo cerca de un dispositivo maestro Bluetooth (un computador o un teléfono móvil) con el que se quiere establecer comunicación. Se puede hacer esto con más de un dispositivo maestro si se desea.

3. El módulo debería ser detectado por ese dispositivo maestro. El software de gestión Bluetooth que esté instalado en este (todos los sistemas operativos incluyen uno "de fábrica") solicitará entonces una clave para poder establecer comunicación con el módulo. La clave por defecto de todos los módulos BlueSMiRF comercializados por Sparkun es "1234". Una vez introducida la clave, el dispositivo maestro reconocerá automáticamente ese módulo como un puerto serie más, cuyo nombre concreto depende del sistema operativo del dispositivo maestro (por ejemplo, en Windows los módulos Bluetooth suelen detectarse como puertos "COMx", en Linux como dispositivos /dev/rfcommX, etc). Afortunadamente, este dato generalmente no será necesario conocerlo.

4. Conectar el módulo a la placa Arduino (es decir, las líneas RX y TX). A partir de aquí ya disponemos de una placa Arduino autónoma capaz de comunicarse utilizando un canal serie con los dispositivos maestros configurados en los pasos anteriores.

5. Para poder enviar datos interactivamente desde el dispositivo maestro a la placa Arduino o bien para observar los datos recibidos de esta en tiempo real, debemos utilizar una aplicación de tipo "terminal serie" como los comentados en el capítulo 3 para sistemas Windows, Linux y Mac OS X (si estamos utilizando como dispositivo maestro un teléfono con Android, una buena aplicación es BlueTerm (https://play.google.com/store/apps). Como siempre, lo único que debemos hacer para utilizar este tipo de programas es seleccionar el puerto serie correspondiente al módulo Bluetooth (de entre los detectados en ese momento, mostrados en una lista) y la velocidad de comunicación (en bits/s) apropiada. Una vez lleguen los datos al dispositivo maestro, otro tema sería qué hacer con ellos (guardarlos en un fichero tabulado, o en una base de datos, o representarlos gráficamente en tiempo real, etc.).

Si las líneas RX y TX del módulo se mantienen conectadas y se desea volver a cargar un nuevo sketch en la placa Arduino, aparecerán problemas de comunicación y el sketch no podrá ser cargado correctamente. Una solución sería desconectar estas dos líneas, cargar el sketch y volverlas a conectar. Otra solución sería utilizar en vez de las líneas RX/TX hardware, un par de líneas SoftwareSerial; de esta manera, la comunicación serie con el módulo Bluetooth no interfiere con la comunicación serie a través de USB.

Ejemplo 8.10: A continuación mostramos un código de ejemplo donde se demuestra lo sencillo que resulta (una vez realizados los pasos anteriores) la gestión de tráfico Bluetooth entre la placa Arduino y un dispositivo maestro. Concretamente, este sketch hace que la placa Arduino envíe inalámbricamente al dispositivo maestro un mensaje infinitas veces, cada dos segundos.

```
#include <SoftwareSerial.h>
//Conectamos el módulo a los pines 2 y 3 (de tipo SoftwareSerial)
SoftwareSerial bt(2,3);
void setup(){
  bt.begin(115200);  //Velocidad por defecto de los módulos BlueSMiRF
}
void loop(){
  bt.println("Hola");
  delay(2000);
}
```

Ejemplo 8.11: El siguiente sketch supone que hemos conectado un LED al pin de salida digital nº 13 de nuestra placa Arduino. Cuando esta reciba a través de una señal Bluetooth (proveniente de algún dispositivo maestro) el carácter “1”, el LED se encenderá; cuando reciba el carácter “0”, el LED se apagará.

```
#include <SoftwareSerial.h>
char carácter;
SoftwareSerial bt(2,3);
void setup() {
  bt.begin(115200);
  pinMode(13, OUTPUT);
}
void loop() {
  //Mientras no se recibe nada, no hago nada
  while (bt.available()==0){;}
  caracter = bt.read();
  if( caracter == '0' ) digitalWrite(13,LOW);
  if( caracter == '1' ) digitalWrite(13,HIGH);
  delay(50);
}
```

Los módulos BlueSMiRF se comercializan con un conjunto de valores “de fábrica” (bits/s por defecto, clave por defecto, nombre del módulo por defecto, etc.) que pueden ser modificados por nosotros si así lo deseamos. Para ello, debemos entrar en el “modo comando” y enviar los comandos de configuración pertinentes.

Esto se hace conectándonos al módulo mediante nuestra aplicación “terminal serie” preferida a una velocidad a 9600 bits/s y escribiendo el comando especial “$$$” (sin las comillas). Atención, esto ha de hacerse dentro del primer minuto después del encendido eléctrico del módulo: después ya no será posible entrar en el “modo comando”. Si se considera este tiempo demasiado corto, se puede modificar y guardar el nuevo tiempo mediante precisamente un determinado comando.

Una vez escrito el comando “$$$”, el módulo debe responder con el mensaje “CMD”. A partir de aquí, antes de enviar cualquier comando, debemos cambiar primero la configuración del programa terminal serie para que los comandos enviados acaben en un carácter de final de línea (el módulo así lo requiere) por lo que deberemos establecer una opción llamada “Newline”, “Carriage return” o similar allí donde antes aparecía la opción “No line ending” o similar, que es la que generalmente está puesta por defecto.

Una vez hecho lo anterior, ya podremos enviar los comandos que deseemos, los cuales pueden ser de tres tipos: comandos para obtener información del módulo, comandos para establecer parámetros de configuración del módulo y comandos para conectar/desconectar con otros módulos. Un comando del primer tipo es por ejemplo “D” (sin comillas), que permite conocer los datos básicos del módulo (bits/s, nombre, dirección, etc.); otro que ofrece información algo más detallada es el comando “E”. Un comando del segundo tipo puede ser el comando “SP,nuevaclave”, que sirve para cambiar la clave del dispositivo (por defecto, “1234”, recordemos) por la especificada. Un comando del tercer tipo puede ser el comando “I,30”, el cual realiza un escaneado durante los segundos especificados (30, en este caso) para detectar la presencia de dispositivos Bluetooth cercanos. Existen multitud más de comandos, pero en este libro no tenemos espacio suficiente para profundizar en ellos; afortunadamente, todos los comandos están perfectamente documentados tanto en el datasheet de los chips RN-41 y RN-42 (de hecho, son los mismos comandos para ambos) como en la estupenda “Advanced User Guide”, descargable de la página web del producto en Sparkfun. Para salir del “modo comando” y volver al modo de transmisión de datos estándar, se ha de escribir el comando “---“ (sin comillas). Recordar establecer la opción “No line ending” otra vez.

Otra placa breakout Bluetooth interesante es la llamada “JY-MCU BT BOARD”, disponible en DealExtreme (entre otros) con código de producto nº 104299. Trabaja por defecto a 9600 bits/s y su clave es también “1234”. Igualmente, dispone de cuatro conectores: “VCC” (a conectar en este caso a 5 V), “GND”, “RXD” y “TXD” y tiene la antena integrada en el PCB. Los pasos a seguir son similares a los ya descritos: cargar el código en la placa Arduino, emparejar el módulo Bluetooth con los dispositivos

maestros a utilizar, conectarlo a la placa y utilizar un programa terminal serie para interaccionar con él.

Una placa breakout Bluetooth curiosa es el "BTBee" de IteadStudio, ya que su forma y la disposición de los pines VCC, GND, RX y TX es compatible con los módulos XBee (y trabajan, como ellos, a 3,3 V). Esta característica hace que sean ideales para cuando tengamos una instalación ya montada con módulos XBee y deseemos realizar un cambio a la tecnología Bluetooth de la forma más rápida y sencilla posible. Incorpora el módulo HC-06, que integra una antena en su PCB. Esta placa se adquiere funcionando por defecto a 38400 bits/s y con una clave de "0000". En la página web del producto hay disponible gran cantidad de información muy recomendable de consultar: desde la referencia de comandos de configuración hasta varios códigos de ejemplo, pasando por diferentes esquemas de conexiones.

Otra placa a destacar es el "LC-05 Bluetooth Serial Module Master/Slave in One" fabricado por AliExpress y distribuido entre otros por ElectronDragon. Tal como su nombre indica, esta placa, a diferencia de las anteriores, puede funcionar tanto en como esclavo (por defecto) como maestro (si se le configura mediante los comandos adecuados). Puede funcionar a 3,3 V o 5 V, trabaja a 9600 bits/s por defecto y su clave es "1234".

Shields que añaden conectividad Bluetooth

Para añadir conectividad Bluetooth, también podemos utilizar shields acoplables a nuestra placa Arduino en vez de módulos independientes. En este caso, la forma de trabajar de los shields es muy similar: a través de dos pines se alimentan eléctricamente (5 V o 3V3 y GND) y a través de otros dos pines se establece una comunicación serie (normalmente de tipo software para no interferir con la comunicación serie hardware existente en los pines 0 y 1). Estos dos pines generalmente pueden ser escogidos dentro de nuestro sketch.

Un ejemplo es el "Bluetooth Shield" de Seeedstudio. Este shield puede actuar tanto como dispositivo maestro como esclavo, por lo que nos puede servir tanto para establecer comunicación con un computador o teléfono móvil (como hemos venido viendo hasta ahora) pero también para establecer comunicación entre dos Arduinos con sendos shields Bluetooth (uno maestro y otro esclavo). Debido a esta versatilidad, su configuración y uso es algo más complejo que lo visto hasta ahora. Hay que tener en cuenta, no obstante, que su rango de detección máximo es de 10 metros (clase 2). En la wiki oficial del producto ofrecen una amplia documentación sobre sus valores por defecto, todas sus configuraciones posibles y diversos casos de

uso. Además, se proporciona una librería propia específica para su manejo y varios códigos de ejemplo. Otros shields muy parecidos al anterior (ya que también pueden funcionar en modo esclavo o maestro) son el “Bluetooth Shield” de Elecfreaks y el “BT shield v2.2” de Iteadstudio, ambos basados en el módulo HC-05, también de clase 2 y con antena integrada.

Hay que distinguir el “BT shield v2.2” del “BT shield v2.1” de Iteadstudio. Este último es un shield que está basado el módulo Bluetooth HC-06 (también de clase 2) pero que tan solo puede funcionar como modo esclavo. Esto hace que la configuración y funcionamiento de este shield sea más sencillo. Concretamente, lo primero que debemos hacer para empezar a utilizarlo es acoplarlo sobre una placa Arduino encendida pero que no esté ejecutando ningún tipo de código. Para asegurar este punto, lo más sencillo es cargar un sketch en el cual las funciones “setup()” y “loop()” estén completamente vacías. Con esto conseguimos alimentar el shield de forma que pueda ser detectado por el dispositivo maestro y pueda ser emparejado (la clave por defecto es “1234”). Una vez reconocido el shield por este dispositivo maestro, debemos desacoplar el shield y cargar seguidamente a la placa Arduino el código deseado (vía USB como siempre). Este código deberá usar el objeto “Serial” para la comunicación serie-Bluetooth (funciona a 9600 bits/s por defecto). Finalmente, hemos de volver a acoplar el shield sobre la placa Arduino y asegurarnos de que el pequeño interruptor que este incorpora está colocado en posición “To Board”.

Si queremos configurar los datos básicos de este shield (nombre, clave, etc.), tendremos que entrar en el “modo comando”. Para ello, deberemos mantenerlo acoplado a la placa Arduino y conectar esta a un computador vía USB, colocando el pequeño interruptor del shield en la posición “To FT232”. Tendremos que volver a cargar un sketch completamente vacío en la placa Arduino y seguidamente, ejecutar nuestra aplicación “terminal serie” preferida a una velocidad de 9600 bits/s para conectarnos al puerto USB (¡no al Bluetooth!): a partir de entonces podremos ejecutar los comandos de configuración que deseemos. Un comando útil es por ejemplo “AT+PINxxxx” (sin comillas), que sirve para cambiar la clave (la nueva se especifica en el lugar de las “x”). Todos los comandos disponibles se pueden consultar en el datasheet del módulo HC-06. Una vez hayamos terminado, deberemos desacoplar el shield y cargar seguidamente a la placa Arduino el código deseado (vía USB como siempre). Finalmente, hemos de volver a acoplar el shield sobre la placa Arduino, asegurándonos de que el pequeño interruptor que este incorpora está colocado en posición “To Board”.

A
DISTRIBUIDORES DE ARDUINO Y MATERIAL ELECTRÓNICO

Existen muchas tiendas online donde podemos adquirir los distintos modelos de placas y shields Arduino. Las variaciones de precio suelen ser de unos pocos euros y el servicio en general es bueno en todas, así que la elección de una u otra se suele basar principalmente en la experiencia personal de cada uno. A continuación, se indican (ni mucho de menos de forma exhaustiva ni completa) alguna de las tiendas más conocidas para adquirir todo lo necesario para nuestros proyectos con Arduino.

El primer sitio donde podemos ir es la tienda oficial de Arduino: http://store.arduino.cc. Además de las placas y shields oficiales, desde aquí podemos adquirir otros elementos, como por ejemplo diferentes shields no oficiales, módulos TinkerKit (por separado o en forma de kit), el microcontrolador Atmega328P suelto pero con el bootloader Optiboot ya cargado, el chip controlador de motores L298N, tarjetas microSD, módulos PoE, cables USB de distinto tipo, ristras de pines-macho y pines-hembra de distinta longitud para colocar en shields, etc. También podemos comprar componentes electrónicos básicos para poder montar cualquier circuito, como resistencias, potenciómetros, LDRs, condensadores, LEDs, pulsadores, cables de distinta longitud, pinzas, breadboards de diferente tamaño, etc.

Otro sitio que podemos consultar es http://arduino.cc/en/Main/Buy, donde podemos ver un listado exhaustivo de los distintos distribuidores de Arduino oficiales reconocidos, clasificados por países. En todos ellos podremos encontrar las diferentes placas y shields oficiales Arduino, así como el resto de componentes necesarios para realizar cualquier proyecto (desde resistencias, potenciómetros,

condensadores, diodos, LEDs, LCDs, transistores, zumbadores, pulsadores, breadboards, cables, multímetros... hasta motores de diversos tipos, material para robótica, módulos TinkerKit, módulos XBee, antenas, diferentes tipos de fuentes de alimentación y sus complementos, teclados numéricos, chips de todo tipo, sensores de toda clase, placas y shields no oficiales...), incluyendo también la posibilidad de adquirir kits completos ya preparados con lo esencial. Algunos de los distribuidores que aparecen en la sección "Spain" son:

Electan	(http://www.electan.com)
CanaKit	(http://www.canakit.es)
Bricogeek	(http://www.bricogeek.com/shop)
Cooking-Hacks	(http://www.cooking-hacks.com)
Ardutienda	(http://www.ardumania.es/ardutienda)
Elect. Embaj.	(http://www.electronicaembajadores.com)
Bcncybernetics	(http://www.bcncybernetics.com)
Ro-botica	(http://www.ro-botica.com/arduino.asp), especializados en robótica.

También existen diversos distribuidores para Argentina, Brasil, Chile, Colombia, Costa Rica, Ecuador, México, Panamá y Uruguay.

Aunque si lo que queremos es consultar durante horas y horas el inmensísimo catálogo que presentan los grandes distribuidores de Arduino y electrónica a nivel mundial (y leer la gran cantidad de interesantísimos artículos y tutoriales que muy a menudo ofrecen sobre sus productos, algunos de los cuales están diseñados y fabricados por ellos mismos), a continuación, se muestra una lista (ni mucho menos exhaustiva) de los más importantes. Algunos de ellos han aparecido a menudo a lo largo del libro.

Sparkfun	(http://www.sparkfun.com)
Adafruit	(http://www.adafruit.com
Makershed	(http://www.makershed.com)

Y también:

DFRobot	(http://www.dfrobot.com)	Freetronics	(http://www.freetronics.com)
Iteadstudio	(http://iteadstudio.com/store)	Seeedstudio	(http://www.seeedstudio.com)
Yourduino	(http://www.yourduino.com)	Fungizmos	(http://store.fungizmos.com)
Modern Device	(http://shop.moderndevice.com)	Cutedigi	(http://www.cutedigi.com)
RSH Electronics	(http://www.rshelectronics.co.uk)	SK Pang	(http://www.skpang.com)
Littlebird Elect.	(http://littlebirdelectronics.com)	Hacktronics	(http://www.hacktronics.com)
Hobbytronics	(http://www.hobbytronics.co.uk)	Mindkits	(http://www.mindkits.com)
Cool Components	(http://www.coolcomponents.co.uk)	Oomlout	(http://www.oomlout.co.uk)

Nkcelectronics	(http://www.nkcelectronics.com)	Eio	(http://www.eio.com)
Australian Robotics	(http://australianrobotics.com.au)	Snootlab	(http://www.snootlab.com)
EarthShine Design	(http://www.earthshinedesign.com)	Kineteka	(http://shop.kineteka.com)
EvilMadScience	(http://evilmadscience.com)	Lees	(http://www.leeselectronic.com)

Proveedores de Arduino que están más específicamente especializados en componentes para robótica son:

Pololu	(http://www.pololu.com)	Solarbotics	(http://www.solarbotics.com)
RobotShop	(http://www.robotshop.com)	RobotBits	(http://robotbits.co.uk)
TrossenRobotics	(http://www.trossenrobotics.com)	SGBotic	(http://www.sgbotic.com)
ActiveRobots	(http://www.active-robots.com)	RobotGear	(http://www.robotgear.com.au)
Toysdownunder	(http://www.toysdownunder.com)	RobotStore	(http://www.robotstore.com)
RobotElectronics	(http://www.robot-electronics.co.uk)	SuperRobot	(http://www.superrobotica.com)
Juguetronica	(http://www.juguetronica.com)		

Otros proveedores dignos de mención también son: Open Electronics (http://store.open-electronics.org), especializados en control remoto y localización, Makerbot (http://store.makerbot.com) y RepRap (http://reprap.org), especializados en el diseño y construcción de impresoras 3D, o Microcontrollers Shop (http://microcontrollershop.com), los cuales venden todo lo relacionado con microcontroladores Atmel y de otras marcas.

Si estuviéramos buscando un componente electrónico tan extraño que no lo encontráramos en ninguno de los sitios anteriores, después de preguntar en nuestra tienda local más cercana aún podríamos consultar los inabarcables catálogos de los siguientes proveedores más importantes a nivel mundial de componentes eléctricos y electrónicos (la lista no es exhaustiva, ni mucho menos), como por ejemplo:

Mouser	(http://es.mouser.com)	Jameco	(http://www.jameco.com)
Farnell	(http://es.farnell.com)	RS	(http://es.rs-online.com)
Digikey	(http://www.digikey.es)	Newark	(http://www.newark.com)
Conrad	(http://www.conrad.com)	Mpja	(http://www.mpja.com)
Maplin	(http://www.maplin.com)	RadioShack	(http://www.radioshack.com)
Verical	(http://www.verical.com)	Vishay	(http://www.vishay.com)
Velleman	(http://www.velleman.eu)	Jaycar	(http://www.jaycar.com)
Satistronics	(http://www.satistronics.com)	Allied El.	(http://www.alliedelec.com)
Future Electronics	(http://www.futureelectronics.com)	Futurlec	(http://www.futurlec.com)
Gateway Catalog	(http://www.gatewaycatalog.com)	BGMicro	(http://www.bgmicro.com)
Sure Electronics	(http://www.sureelectronics.net)	Chip1Stop	(http://www.chip1stop.com)
BPE Solutions	(http://www.bpesolutions.com)	DealExtreme	(http://www.dx.com)
Goldmine	(http://www.goldmine-elec.com)	RapidOnline	(http://www.rapidonline.com)
AllElectronics	(http://www.allelectronics.com)	PartsExpress	(http://www.parts-express.com)

OnlineComponents (http://www.onlinecomponents.com)

Por si fuera poco, también disponemos de dos páginas muy prácticas que nos pueden ayudar a buscar dónde venden un componente determinado y comparar su precio (y stock actual) entre distintos proveedores: http://www.findchips.com y http://www.octopart.com.

Kits

De todas formas, lo más cómodo para el principiante de Arduino es adquirir los llamados "kits", formados por una placa Arduino (normalmente, el modelo UNO) más una colección relativamente completa de componentes electrónicos ya preempaquetados. Incluso algunos de ellos vienen con un libro didáctico de proyectos paso a paso. Adquiriendo uno de estos kits, pues, el usuario no se tiene que preocupar buscando individualmente cada elemento que necesita para sus proyectos porque la mayoría de ellos ya los habrá adquirido de golpe con el kit. La mayoría de proyectos que vistos en este libro se pueden hacer realidad con cualquiera de estos kits que se listan a continuación:

"Arduino Starter Kit" (de la tienda oficial de Arduino): contiene resistencias de diferentes valores, un termistor, un LDR, un potenciómetro, un diodo, condensadores de diferentes clases, LEDs de diferentes colores, pulsadores, cables, una breadboard, pines para conectar a shields, un transistor bipolar NPN y otro MOS, un zumbador, un chip L293D, un motor DC de 6/9 V y un servomotor, un par de sensores de inclinación, un par de optoacopladores y un cable USB A/B para conectar el computador a la placa. Además, incorpora un libro de 170 páginas donde se desarrollan 15 proyectos didácticos, de más sencillos a más complicados; la lista completa se puede leer en http://arduino.cc/en/Main/ArduinoStarterKit. Existen dos variantes de este kit: uno que incorpora además de todo lo anterior una placa Arduino UNO y otro en el que no.

"Getting started with Arduino kit" (de Makershed): contiene resistencias de diferentes valores, un par de LDRs, un breadboard, cables, pulsadores, LEDs de diferentes colores, un contenedor para pilas de 9 V con conector de 2,1 mm, un cable USB y la placa Arduino UNO. Otro kit de Makershed más completo es el **"Ultimate Microcontroller Pack"**, que contiene además de todo lo anterior, dos motores mini-servos, un minimotor DC, un motor de vibración, una pantalla LCD de 16x2, una resistencia sensible a la fuerza (FSR), una caja para guardar los componentes, más modelos de breadboards,

condensadores, sensor de inclinación, pulsadores, termistores, interruptores, un diodo, un transistor bipolar NPN, un altavoz, un zumbador y varias ristras de pines.

”Inventor's kit” (de Sparkfun): contiene resistencias de diferentes valores, potenciómetros de varios tipos, un LDR, diodos, LEDs de diferentes colores, pulsadores, un pequeño motor DC, un motor mini-servo, dos transistores bipolares NPN, un relé, un sensor de temperatura, un sensor de flexión, un zumbador, cables, ristra de pines, un breadboard y un soporte para fijarla, un cable USB y la placa Arduino UNO. Contiene además una guía ilustrada de proyectos. Un kit parecido al anterior es el “**Starter kit - Flex**”, el cual contiene además un termistor pero no incorpora ni la guía ilustrada, ni los transistores ni los diodos ni los motores ni el relé ni el sensor de temperatura, entre otras diferencias menores. Por otro lado, Sparkfun también distribuye el “Jumper Wire Kit”, que simplemente es un conjunto extra de cables de diferentes longitudes, ideal para tenerlos siempre a mano por si faltan en nuestros proyectos de prototipado.

”Starter pack” (de Adafruit): contiene resistencias de diferentes valores, potenciómetros, un LDR, LEDs de diferentes colores, pulsadores, cables, un breadboard pequeña, una Protoshield, un adaptador AC/DC a 9 V, un contenedor para pilas de 9 V con conector de 2,1 mm, un cable USB y la placa Arduino UNO. Un kit parecido al anterior es el **“Budget pack”**, pero éste no contiene ni el contenedor para pilas ni el adaptador AC/DC ni el Protoshield, entre otras diferencias menores.

”ARDX Experimentation kit” (de Adafruit): contiene resistencias de diferentes valores, potenciómetros, una resistencia sensible a la fuerza (FSR), un LSR, dos diodos, LEDs de diferentes colores, pulsadores, un pequeño motor DC, un motor mini-servo, dos transistores bipolares NPN, un relé, un sensor de temperatura, un zumbador, cables, un breadboard y un soporte para fijarla, un clip para unir una pila de 9 V a un jack macho de 2,1 mm, un cable USB y la placa Arduino UNO. Contiene además una guía ilustrada de proyectos.

Existen muchos otros kits; dignos de mención son: el **“Kit Workshop”** de Ardumania (cuyos componentes más destacados son un sensor de temperatura, un LDR, un motor mini-servo, un zumbador, un transistor MOSFET y un diodo, pero ojo, no incluye ni el cable USB ni la placa); el **“Arduino Starter Kit”,** el **“Arduino Sidekick Basic Kit”** y el **“Arduino Lab Kit”** de Cooking-Hacks (a cual más completo, además del

“Components Kit” que viene sin Arduino); el **“Starter Kit”** de Cutedigi (el cual puede ser adquirido completo o bien por partes llamadas “A”,”B”,”C”, etc); el **“Arduino Educational Kit”** de Hacktronics; el **“Low-cost Starter Set”** de Yourduino; el **“Starter Kit”** de EarthShine (que contiene un manual de proyectos muy bien escrito, al igual que el **“Arduino Workshop”** de SmileyMicros.com); los **“Kits de iniciación”** de TodoElectronica.com, el “**Electronics Tools and Parts Starter Kit**” de CuriousInventor (sin Arduino), los **“Electronics Components Packs 1a”** y **“2a”** de Makershed (sin Arduino), los **“Component Starter Kit”** y **“Resistor Starter Kit”** de Akafugu (sin Arduino), etc.

Si nos interesa también el ámbito de los kits educativos de electrónica general (sin relación con Arduino), son interesantes las páginas web http://www.elektor.es y la mencionada http://www.todoelectronica.com, las cuales también son responsables de publicaciones periódicas didácticas sobre electrónica muy interesantes.

B

CÓDIGOS IMPRIMIBLES DE LA TABLA ASCII

Código numérico	Carácter	Código numérico	Carácter	Código numérico	Carácter
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	´	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l

45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_	127	DEL
64	@	96	`		

Si se quiere conocer la tabla ASCII al completo (y sus diferentes ampliaciones), se puede consultar http://www.asciitable.com. En sistemas Linux también podemos visualizarla mediante el comando `man ascii`.

C

RECURSOS PARA SEGUIR APRENDIENDO

Una vez finalizada la lectura de este libro, ¡esto no ha hecho más que empezar! Las posibilidades que se abren a partir de ahora son realmente casi infinitas. No obstante, el lector necesitará otros recursos de información para progresar. Así pues, para evitar el estancamiento, a continuación, ofrecemos una lista de recursos recomendados (tanto en papel como en formato web) para profundizar en el conocimiento de las diversas áreas tratadas en el libro.

Plataforma Arduino

El mejor sitio para continuar aprendiendo todo sobre Arduino es, lógicamente su página web oficial. Lo más interesante es tal vez su "Playground" (http://www.arduino.cc/playground), wiki editable por la comunidad donde se muestran multitud de trucos y tutoriales prácticos de la más diversa índole. Otra fuente de información (aunque mucha de ella ya ha sido expuesta en este libro) es el apartado de "Hacking" de la misma web oficial (http://www.arduino.cc/en/Hacking). También existe, dentro de la web oficial, un apartado a enlaces externos con multitud de información sobre Arduino (http://www.arduino.cc/en/Tutorial/Links).

La ayuda que podemos recibir no se acaba aquí, ni mucho menos: también podemos formar parte de la amplia comunidad Arduino existente en el mundo a través del foro oficial (http://forum.arduino.cc). Desde allí podremos ayudar y ser ayudados en cualquier tema relacionado con Arduino: existen muchas categorías,

que van desde consultas sobre electrónica básica, programación básica, microcontroladores, problemas en la instalación y uso del IDE, tipos de sensores, tipos de actuadores, maneras de comunicar la placa –red, serie, etc.–... hasta consultas relacionadas con ámbitos de uso, como la robótica, la domótica, el diseño interactivo, educación, ciencia, uso de textiles... pasando por consultas sobre posibles mejoras al hardware y al software Arduino, noticias sobre eventos y talleres, etc. Incluso existe un apartado específico para la comunidad hispanohablante. Es decir, resumiendo: cualquier persona interesada en Arduino debe consultar el foro oficial si quiere acceder a una ingente cantidad de información de valor incalculable, aportada gracias a la cooperación desinteresada de toda la comunidad, que hace que Arduino sea un proyecto cada vez más y más grande. ¡No olvides colaborar y aportar tu propio granito de arena!

Otro sitio de reunión de la comunidad Arduino hispanohablante es el portal Arduteka (http://www.arduteka.com), útil para estar al día de las novedades y acceder a completos tutoriales.

No está de más comentar también la existencia tanto de la cuenta de Twitter oficial de Arduino (@arduinoteam) como del blog oficial de Arduino (http://blog.arduino.cc), que nos permitirán estar al día de los proyectos más interesantes que surjan dentro de la comunidad, así como de las novedades oficiales que vayan apareciendo.

Finalmente, en la dirección http://www.arduino.cc/en/Main/ContactUs se listan las direcciones de correo electrónico utilizadas por el Arduino Team para recibir sugerencias o consultas sobre algún aspecto de Arduino que no haya quedado suficientemente aclarado en las páginas anteriores.

Respecto a la información bibliográfica existente sobre Arduino, es un hecho que cada vez es más extensa y variada. Un rápido vistazo al catálogo de la librería online Amazon, por ejemplo nos revelará multitud de títulos: algunos más enfocados a la vertiente hardware de Arduino y otros más a la vertiente de programación, otros centrados en ser un compendio de proyectos y otros en ser meras introducciones, otros más especializados en el estudio de la comunicación de sensores en redes inalámbricas y otros en la comunicación con sistemas Android, otros relacionados con la robótica y otros más en la domótica, o en el textil electrónico, etc. En este sentido, para poder elegir el libro que realmente necesitemos, recomiendo consultar su índice y así confirmar si nos ofrece lo que buscamos. También es importante fijarse en la fecha de edición, ya que fechas demasiado antiguas incorporarán códigos obsoletos.

Electrónica general

Para aprender electrónica, la mejor opción es adquirir un libro especializado en la materia. Para iniciarse recomiendo los siguientes:

*”Make: Electronics” Charles Platt. *Make; 1ed (2009)*
*”Getting Started In Electronics” Forrest M. Mims III. *Master Publishing, Inc. (2003)*
*“Electronics for Dummies” Cathleen Shamieh. *For Dummies;2ed (2009)*

De un nivel intermedio podemos destacar:

*”Complete Electronics Self-Teaching Guide” Earl Boysen. *Wiley;3ed (2012)*
*”Practical electronics for inventors” Paul Scherz. *McGraw-Hill; 2ed (2006)*
*”Circuitbuilding for dummies” H.Ward Silver. *For Dummies; 1ed (2008)*
*”Electronic projects for dummies” Earl Boysen. *For Dummies; 1ed (2006)*
*”Teach Yourself Electricity and Electronics” Stan Gibilisco. *McGraw-Hill; 5ed (2011)*
*“Beginner's Guide to Reading Schematics” Robert J. Traister.*TAB Books; 2ed (1991)*
*”Lessons in electric circuits” Tony R. Kuphaldt *(2006-2010)*

El último libro mencionado en realidad son varios volúmenes descargables gratuitamente de http://openbookproject.net/electricCircuits Viene acompañado de un conjunto de preguntas, disponibles en http://www.ibiblio.org/kuphaldt/socratic.

Y de un nivel bastante más avanzado:

*”The Art of Electronics” P. Horowitz. *Cambridge U.P;2ed (1989)*
*“The Circuit Designer’s Companion” Tim Williams. *Newnes;2ed (2005)*
*”Practical Introduction to Electronic Circuits” M.H.Jones. *Cambridge U.P;3ed (1996)*

Si, no obstante, se desea utilizar recursos online, una página web que contiene tutoriales de electrónica y circuitos de ejemplo interesantes es http://www.kpsec.freeuk.com, y también http://www.radio-electronics.com. Una web que contiene muchos ejemplos de circuitos es http://www.extremecircuits.net. Si se quieren conocer más recursos online, una web con muchos enlaces es http://www.dapj.net/hobby/educational.

Proyectos

Existen varios portales web que diariamente recopilan proyectos muy interesantes publicados en blogs personales. En realidad, muchos proyectos no están

relacionados exclusivamente con Arduino, sino que muestran proyectos autónomos de robótica, domótica, control remoto, multimedia, impresión 3D…, pero consultar estos portales es una buena manera para estar al día de las ideas que van surgiendo en la comunidad y para tomar buena nota de los trucos y conocimientos empleados por mucha gente. Está claro que para aprender procedimientos frescos y adquirir ideas nuevas, lo mejor es estudiar proyectos de otras personas, y estos portales nos lo permiten. Algunos de ellos son:

Instructables (http://www.instructables.com)
Makezine (http://blog.makezine.com/arduino)
MakeProjects (http://makeprojects.com/Topic/Arduino)
HackADay (http://hackaday.com/category/arduino-hacks)
HackNMod (http://hacknmod.com)
Dangerous Prototypes (http://dangerousprototypes.com)
Electronics Lab (http://www.electronics-lab.com/blog)
BricoGeek (http://www.bricogeek.com)
Embedds (http://www.embedds.com)
Sección de proyectos Jameco (http://www.jameco.com/Jameco/PressRoom/diy.html)
Sección de proyectos Arduino (http://arduino.cc/playground/Projects/ArduinoUsers)
Proyectos del HLT (http://hlt.media.mit.edu/?cat=20 , http://hlt.media.mit.edu/?p=1283 y ?p=1314)

Algunos distribuidores (como Sparkfun, Adafruit, Makershed, Bricogeek, Cooking-Hacks, y más) incorporan dentro de sus páginas oficiales secciones donde informan puntualmente de las novedades y los eventos más destacables dentro del mundo de la computación física, siendo pues su consulta muy recomendable para estar al día en este ámbito.

ÍNDICE ANALÍTICO

C

Ch

C

D

E

F

G

H

I

J

L

M